University of Bergen
Department of Informatics

Parallel Community Detection in Incremental Graphs

Author: Magnus Tønnessen
Supervisors: Fredrik Manne, Johannes Langguth

UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

June, 2023

Abstract

The problem of community detection in large, expanding real-world networks presents significant challenges due to the scale and complexity of these networks. Traditional algorithms struggle to provide optimal solutions or require unviable computational resources. In this thesis, we address these challenges by exploring, designing and evaluating parallel computing strategies for community detection in incremental graphs. We provide a novel parallel implementation of the NCLiC algorithm by dividing its phases into parallel tasks using a shared memory approach. The algorithm has been extensively tested on various graphs. The results demonstrate promising performance improvements and scalability while retaining the quality of the partitions. The parallel implementation of the Leiden algorithm used for pre-clustering shows virtually no loss in modularity and obtained speedups up to a factor of 10.3. The refinement and merging phases of the parallel NCLiC algorithm obtained speedups up to 18.42 and 10.36, respectively, resulting in a total speedup of up to a factor of 6.73.

Acknowledgements

I would like to express my deepest gratitude to my supervisors, Fredrik Manne and Johannes Langguth, for their unwavering support and guidance throughout the entire duration of my master's thesis. Their expertise, patience and invaluable feedback have been instrumental in shaping this research work.

I would also like to thank my fellow student for the discussions, collaborations and shared experiences, and my better half and family for their persistent encouragement and for putting up with me. Their belief in my abilities has been a constant source of motivation.

`Magnus Tønnessen`
Thursday 1st June, 2023

“The way the processor industry is going, is to add more and more cores, but nobody knows how to program those things. I mean, two, yeah; four, not really; eight, forget it.”

– Steve Jobs, Apple

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Chapter 1

Introduction

1.1 Problem

As society delves further into the epoch of the Information Age, we are witnessing an increasing number of large real-world networks. These complex structures have become omnipresent, from social networks and web graphs to biological systems and transportation networks, representing the essence of interconnectedness in various domains. Along with the explosive growth of data, many of these networks are not static but are continuously expanding in size and complexity day by day. The sheer enormity of the networks is forcing researchers to reconceive traditional methods of data analysis and pattern discovery, focusing on methods that can scale and adapt to the increasing demands of growing networks.

Community detection is one such approach that offers insightful information about graphs. Communities, also often referred to as clusters, are defined as groups of vertices with denser connections within the group than with the rest of the graph. Unveiling these communities can yield valuable information about the underlying structure of the graphs, revealing their organisational principles and functional mechanisms. Community detection, therefore, has significant implications for various fields, including sociology, biology, computer science and physics, where it has been applied to areas such as unveiling hidden graph properties, predicting future graph growth and optimising resource allocation.

However, the process of community detection in ever-growing graphs introduces significant challenges. Traditional algorithms struggle to keep up with the scale and complexity of contemporary graphs, often leading to sub-optimal solutions or unviable computational demands. In response to these challenges, there is a pressing need to explore more efficient algorithmic strategies that can scale effectively to large and growing graphs.

Parallel computing emerges as a promising solution to this scalability issue. By dividing the computational tasks and distributing them across multiple processors, parallel computing allows for simultaneous data processing, enabling an approach to significantly reduce computation times and improve performance. In the context of community detection, leveraging parallel computing can lead to more efficient and scalable algorithms capable of handling large, complex graphs with enhanced precision and speed.

The focus of this thesis is to explore, design, and evaluate parallel computing strategies for community detection in incremental graph; graphs generated by continuously arriving chunks of edges. We propose a novel parallel implementation of the NCLiC algorithm and establish some requirements for the algorithm, to minimise the loss in quality and fully utilise the available resources. Through extensive testing, we analyse the performance of the algorithm in terms of efficiency and quality. Our parallel implementation shows promising results in terms of the quality of the communities detected, efficiency and scalability.

1.2 Thesis Outline

- **Chapter 1** Presents the problem and the goal of the thesis
- **Chapter 2** Presents the theoretical background of the thesis, including network science, community detection, incremental graphs and parallelism
- **Chapter 3** Presents existing community detection algorithms (Louvain, Leiden, NCLiC) relevant to this thesis
- **Chapter 4** Presents related parallel algorithms and research
- **Chapter 5** Presents and discusses our implementation of the parallel NCLiC algorithm
- **Chapter 6** Presents and discusses the results of the tests of the algorithm
- **Chapter 7** Conclusion and future work

Chapter 2

Background

In this chapter, we present the theoretical background for this thesis, including network science, community detection and modularity, incremental graphs and parallelism. We also discuss the choice of programming language used to implement the parallel algorithm.

2.1 Network Science

Network science is the study of large and complex networks, such as social networks, biological networks, transportation networks, and information networks [22]. This field aims to understand how the structure of a network influences its behaviour and how the interactions between its components shape the network's properties. Its roots go back to Königsberg in 1735 when the Swiss mathematician Euler proved that the well-known problem *The Bridges of Königsberg* could not be solved [13]. With new types of networks continuously appearing, such as information and social media networks, the field is more relevant than ever.

A graph is a data structure consisting of a set of vertices and a set of edges that connect the vertices. The vertices represent the entities in the graph, for example, people in a social network, neurons in the brain, or websites on the internet. The edges represent the relationships between these entities, such as friendships between people, synaptic connections between neurons, or hyperlinks between websites.

A weighted graph G is defined as a triple $G = (V, E, w)$, where V is the set of vertices, E is the set of edges and w is a weight function that assigns a weight to each edge. The edges in a weighted graph are typically represented as pairs of vertices, where each pair represents a connection between two entities. The weight of an edge represents the strength of the relationship or the distance between the entities it connects. The open neighbourhood of a vertex v is the set of vertices that share an edge with v, formally defined as $N_G(v) = \{u \mid \{u, v\} \in E(G)\}$.

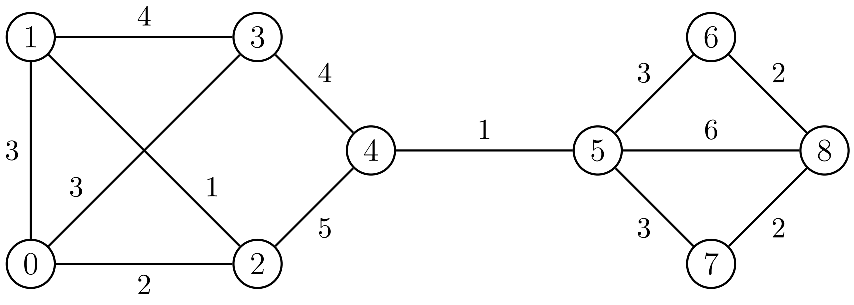


Figure 2.1: Example of a weighted graph

Figure 2.1 shows an example of a graph. The graph has nine vertices, labelled from 0 to 8, and 13 edges, with different weights. The graph can be formally defined as $G = (V, E, w)$ where V is the set of vertices 0 to 8, and E is the set of edges, where each edge is represented $\{i, j\}$ if there is an edge between vertices i and j and w is the weight function. An example of an open neighbourhood is $N_G(4) = \{2, 3, 5\}$.

A real-world example that could be modelled as a weighted graph is a road network, where each vertex represents a city, and each edge represents a road connecting two cities. Weights can be assigned to the graph's edges to reflect the distance between the cities or the time it takes to travel between them.

The structure of a graph can be described using various measures such as degree distribution, centrality and clustering coefficient. Degree distribution [8] refers to the pattern of connections between the entities in a graph, while centrality [6] measures the importance of a vertex within a graph and can be used to identify critical vertices. The clustering coefficient [33] measures the tendency of vertices in a graph to cluster together.

2.2 Community Detection

Community detection is the process of identifying sets of vertices within a graph that are more densely connected to each other than to vertices in other communities. It is a fundamental problem in network analysis on applications in various fields such as social networks, biology, and transportation systems. Community detection aims to uncover a graph's underlying structure and provide insights into its properties. Figure 2.2 shows how the graph in Figure 2.1 can be partitioned into two communities indicated by the colouring of the vertices. The partition in the figure can be defined as $\mathcal{P} = \{\{0,1,2,3,4\},\{5,6,7,8\}\}$.

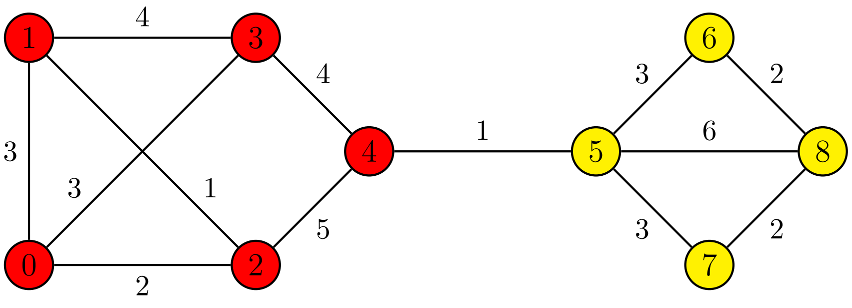


Figure 2.2: Example of a graph with communities

2.2.1 Modularity

The problem of community detection is often formulated as an optimisation problem. The goal is to maximise a quality function that measures the strength of the community structure, subject to certain constraints. The most common quality function in community detection is modularity, introduced by Newman and Girvan in 2004 [26]. Modularity measures the deviation of the number of edges within communities from what would be expected in a random graph with the same degree sequence [18]. The modularity score for unweighted and undirected graphs lies in the range $[-0.5, 1]$. A positive modularity score indicates a graph with a community structure denser than expected in a graph with randomly distributed edges, while a negative score indicates a sparser structure.

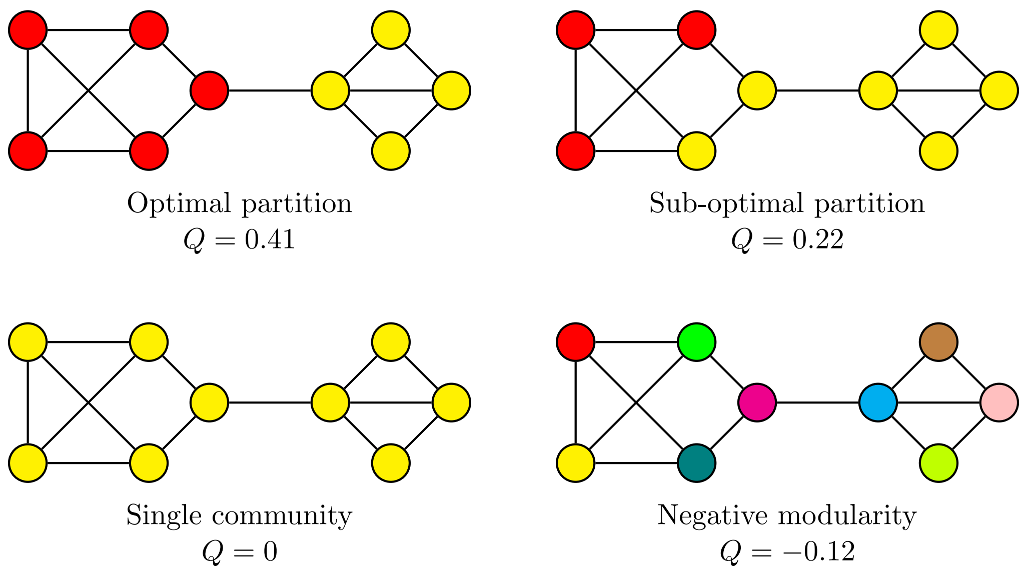
The modularity Q of a partition of a graph into communities is defined as

$$Q = \frac{1}{2m} \sum_{i,j \in V(G)} \left(A_{ij} - \gamma \frac{k_i k_j}{2m} \right) \delta(c_i, c_j)$$

where

- A_{ij} is the weight of the edge between vertices i and j
- $\gamma \in [0, 1]$ is the resolution parameter, which sets a lower limit for the density of the communities
- m is the total weight of the edges in the graph
- k_i and k_j are the sums of the weights of the edges incident to vertices i and j, respectively
- c_i and c_j are the communities that vertices i and j belong to
- $\delta(c_i, c_j)$ is the Kronecker delta which takes the value 1 if $c_i = c_j$, i.e. vertices i and j belong to the same community, and 0 otherwise

Figure 2.3 depicts how different partitions will affect the modularity score for the graph in Figure 2.1. In the optimal partition, only one edge connects the communities, whereas, in the sub-optimal partition, the communities share 3 edges. The partition with only one community results in a modularity score of 0, and with all vertices in singleton communities, the modularity is negative.


Optimal partition
$Q = 0.41$

Sub-optimal partition
$Q = 0.22$

Single community
$Q = 0$

Negative modularity
$Q = -0.12$

Figure 2.3: Different modularity scores for the graph in Figure 2.1

Although modularity is a widely used measure for community detection in complex graphs, it has some weaknesses that can affect performance. One weakness of modularity is its resolution limit, which refers to the inability to detect smaller communities embedded within larger ones [15]. Real-world networks often consist of many small communities rather than a few large ones, meaning the resolution limit may have significant practical implications. Modularity also tends to produce communities of similar size, which may not reflect the actual structure of the network, where some communities are naturally larger or smaller.

Finding the partition of a graph that results in optimal modularity is NP-complete [7], which means that finding the exact solution is computationally intractable for large graphs. Therefore, various heuristics have been proposed to compute good solutions. Some approaches include simulated annealing [16], extremal optimisation [12], hierarchical agglomeration [9] and spectral algorithms [25]. Each has its strengths and weaknesses, and the choice of which one to use will depend on the specific characteristics of the graph being analysed and the research question at hand. In this thesis, we mainly focus on two heuristics for community detection in static graphs: the well-known Louvain algorithm and the more recent Leiden algorithm, explained respectively in Sections 3.1 and 3.2.

2.3 Incremental Graphs

Data streams are sequences of data elements generated and processed continuously over time [17]. They are characterised by their infinite length, dynamic nature, and unbounded data volume. Real-world examples of streaming data include computer logs, stock prices and social media events. Such services generate vast amounts of data we often want to process and analyse instantly. However, if instant data processing is not crucial, we can let the data pile up in a chunk, decided by a size limit or time frame. Processing chunks of data may speed up the process and improve the quality of the computations.

Incremental graphs are graphs generated from chunks of data and constantly change as new vertices and edges are added or existing vertices and edges are deleted. The study of dynamic graphs is an active area of research, and computations of interest include, among others, finding a minimum spanning tree, predicting the probability of a link appearing or, in our case, detecting communities. Methods designed for static graphs can be extremely computationally expensive to apply repeatedly for each new chunk of data that arrives. Thus, methods specifically developed to analyse incremental graphs are needed.

Community detection is already a complex process when performed on static graphs where all information is known beforehand. It becomes even more complex when applied to incremental graphs, where new information arrives continuously. When static algorithms are used to detect communities in an incremental graph, the asymptotic running time of the algorithm depends on the size of the entire graph and increases every time new data arrives. Hence, the processing time of repeatedly reclustering the whole graph eventually exceeds the interval at which the data arrives. Therefore, an algorithm designed for community detection in incremental graphs would have to depend solely on the size of the arriving data. In this thesis, we focus on such an algorithm designed for community detection in incremental graphs, called the NCLiC algorithm, explained in Section 3.3.

2.4 Parallelism

As problems become computationally harder and the instances become larger, the need for effective algorithms increases. Parallelism is a fundamental concept in computing and refers to the simultaneous execution of multiple tasks or processes within a computer system. In its essence, parallel programming is about dividing computational tasks into smaller, independent units that can be executed concurrently. This can be achieved at several levels, including data, task, and instruction level parallelism. Data parallelism refers to the simultaneous processing of large data sets, often using vector or array operations. Task parallelism involves the concurrent execution of different tasks, while instruction parallelism relates to executing multiple instructions from a single instruction stream simultaneously.

Common paradigms of parallel programming include shared memory, distributed memory, and hybrid models. In the shared memory paradigm, all processes share a common address space, and communication between them occurs via reading and writing to shared variables. In the distributed memory paradigm, each process has its own private memory. Processes communicate by sending and receiving messages.

Parallel programming can offer substantial benefits. The most significant advantage is speed. By dividing a problem into smaller parts that can be solved concurrently, parallel programming can significantly reduce the time required to process large data sets or solve complex computational problems. This is particularly beneficial in fields like network science, where handling large-scale networks can be computationally expensive.

However, parallel programming also brings challenges. Designing parallel algorithms can be complex, requiring careful synchronisation of tasks to avoid issues such as race conditions [24] or deadlocks [10]. Furthermore, the overhead involved in coordination and communication between tasks can sometimes offset the benefits of parallelism, particularly for small-scale problems or when the tasks are not entirely independent. Moreover, not all problems are inherently parallelisable, known as Amdahl's law [1], meaning that there's a limit to the speedup that can be achieved by adding more processing elements. The scalability of parallel programs is also an essential factor, as efficient parallel programs should maintain performance gains as the number of processors increases.

Another aspect that is crucial for effective parallel programming is selecting the right programming language. A programming language optimised for parallelism can harness the full potential of multi-core processors and distributed systems, improving performance and scalability. By carefully considering the language, developers can maximise performance, scalability, and the efficient execution of concurrent tasks.

2.5 Programming Language

Two programming languages were considered to implement the algorithms used in this thesis. C++ has long been the preferred language for high-performance computing due to its ability to produce low-level code and optimise memory and processing resources. It is a high-level, general-purpose programming language introduced by Stroustrup in 1985 [28] and was designed as an extension to C, incorporating object-oriented programming and other features to improve performance and flexibility. The language's extensive history has led to vast knowledge, libraries and tools developers can leverage when working on complex concurrent systems. However, C++'s manual memory management and complex syntax can make it difficult to write safe concurrent code and may result in subtle and hard-to-find bugs.

Rust, on the other hand, is a more recent language, developed by Hoare in 2010 [20] and backed by Mozilla Research, with the first stable release in 2015. Rust was developed with performance, memory safety and concurrency as a foundation, making it particularly well-suited for high-performance computing. The language aims to be as efficient and portable as C++ without sacrificing safety, and since it does not perform garbage collection, Rust is often faster than other memory-safe languages [3]. In the following sections, we compare parallelism in Rust and C++, and explain which synchronisation mechanisms Rust provides.

2.5.1 Rust

The most conspicuous feature in Rust is how the language manages memory with the so-called ownership model. The ownership model enforces a set of strict rules on resource access and sharing that the compiler checks.

```
 1 fn function_1(s: String) -> String { return s; }
 2 fn function_2(s: &String) -> usize { return s.len(); }
 3 fn function_3(s: &mut String) { s.push_str(", World!"); }
 4
 5 fn main() {
 6   let s1 = String::from("Hello");
 7   let mut s2 = function_1(s1);
 8
 9   println!("s1: {}", s1);
10   -> use of moved value:  ‘s1‘
11
12   let length = function_2(&s2);
13
14   let s3 = &mut s2;
15   function_3(s3);
16
17   let s4 = &mut s2;
18   -> cannot borrow ‘s2‘ as mutable more than once at a time
19   let s5 = &s2;
20   -> cannot borrow ‘s2‘ as immutable because it is also borrowed as mutable
21 }
```

Listing 2.1: Example of how the ownership model works with compiler errors

Listing 2.1 shows three different methods for accessing variables in Rust and compile-time errors that occur when the rules of the ownership model are broken. The keyword `mut` defines a variable as mutable. Variable `s1` in line 6 is immutable, which means that a new value cannot be reassigned to `s1`. Variable `s2` in line 7 is defined as mutable and can be assigned a new value later in the execution.

The first method to access a variable is to give ownership of the value to another variable. In line 7, the ownership of the value in variable `s1` is given to `function_1`. At this point, `s1` is no longer a valid variable because it does not own a value and cannot be used further. When `function_1` returns the value, the ownership is given to variable `s2`. In line 9, the program tries to print the value of `s1`, which is detected during compile time and will result in an error message.

The second method is to give an immutable reference to the value, called borrowing, which is denoted with an ampersand `&`. In line 12, an immutable reference to variable `s2` is borrowed to `function_2`, meaning the function cannot change the value, only read it, even though variable `s2` is defined as mutable. With the immutable reference given to `function_2`, it reads the value and returns the length of the string.

The last method is to give a mutable reference to the value. In line 14, variable `s3` borrows a mutable (`mut`) reference (`&`) to variable `s2`, which is then passed to `function_3`. This means that `function_3` can change the value of `s2`. When `function_3` is done executing, variable `s2` will have the value `"Hello, World!"`.

An important rule that is enforced by the ownership model is that there can either exist one mutable reference or one or more immutable references to the same variable simultaneously. In line 17, the compiler will give an error message because variable `s3` has already borrowed a mutable reference to `s2`. For the same reason, variable `s5` cannot borrow an immutable reference to `s2` in line 19.

The ownership model in Rust ensures memory safety and completely avoids data races by ensuring that all references point to valid data and that mutable and immutable references cannot exist simultaneously. This feature is essential when working with concurrency because memory can only be altered by one thread or read by multiple threads. Another outcome of how the ownership model keeps control of the references to variables is that Rust does not need a garbage collector. When a variable goes out of scope, the rules of the ownership model guarantee that no references to the variable exist. Hence, the allocated memory can be freed instantly. Since it does not perform garbage collection, Rust is often faster than other memory-safe languages [3]. The ownership model helps developers avoid common memory management pitfalls and often discovers potential bugs during compile time, resulting in safer and more reliable concurrent code. Understanding run-time errors is often more manageable because of the absence of several potential bugs in compiled programs.

2.5.2 Concurrency in Rust and C++

C++ and Rust offer different approaches to concurrency, and the use case depends on the specific needs of the project. In this thesis, we worked with shared memory, which means that memory safety and avoiding race conditions were in focus. This section compares the concurrency mechanisms available in both languages, focusing on the OpenMP library in C++ and the Rayon library in Rust, comparing their features, similarities, and differences to provide an understanding of their concurrency models.

OpenMP is an application programming interface (API) that supports shared-memory code parallelisation in C++, C, and Fortran. The API uses compiler directives to execute parallel code, simplifying the code needed to introduce parallelisation. The simple syntax needed to distribute tasks among available processors in a system easily has made it an attractive choice for parallel programming in C++. OpenMP has been widely adopted in high-performance computing and scientific applications, where its ease of use, portability, and performance-tuning capabilities are highly valued.

Rayon is a parallelism library for Rust that utilises a work-stealing algorithm to distribute tasks among threads effectively. Rayon provides an intuitive, well-documented API that allows developers to execute parallel tasks easily without manual thread management or synchronisation. Combined with the ownership model, the parallel iterators and scoped thread pools provided by Rayon make it easy to implement safe, concurrent code. Additionally, Rayon's focus on safety and zero-cost abstractions aligns with Rust's core principles, making it a popular choice for concurrency in Rust.

OpenMP and Rayon provide simple, high-level abstractions for parallel programming, allowing developers to implement concurrency with minimal code changes. They both support dynamic task scheduling, which helps in efficiently distributing tasks among available threads. However, there are some key differences between the two libraries. While OpenMP employs a shared-memory model, Rayon is designed around Rust's ownership system, focusing on safety and avoiding data races. This makes Rayon's approach to concurrency more predictable and less prone to bugs.

In terms of performance, OpenMP's maturity and widespread use in high-performance computing might give it an edge in certain applications. However, Rayon's work-stealing algorithm and focus on safety make it a strong contender in the Rust ecosystem. The choice between OpenMP and Rayon ultimately depends on the specific use case and the desired trade-offs between performance, safety, and ease of use.

```
1 #pragma omp parallel for reduction(+:sum)
2 for (int i = 0; i < N; i++) {
3   if (arr[i] % 2 == 0) {
4     sum += arr[i] * arr[i];
5   }
6 }
```
Listing 2.2: Parallel iteration in C++

```
1 let sum = arr
2   .par_iter()
3   .filter(|n| n % 2 == 0)
4   .map(|n| n * n)
5   .sum();
```
Listing 2.3: Parallel iteration in Rust

Listings 2.2 and 2.3 shows examples of how a set of operations can be applied to all elements in an array with a parallel for loop in C++ and with a parallel iterator in Rust. The code calculates the sum of all even squares between 0 and a number `N`.

In line 1 in Listing 2.2 the compiler directive `#pragma omp parallel for` tells the compiler that the line that follows is a for loop that should be executed in parallel. The `reduction(+:sum)` part specifies that the reduction operation should be performed, doing a summation of the elements, into the variable `sum`.

Listing 2.3 shows how the same operation can be done in Rust. Data parallelism like this is usually handled with parallel iterators. By calling the method `par_iter` from the Rayon library, Rust converts the array into a parallel iterator. The chained methods `filter`, `map` and `sum` will then be applied to each element in parallel, and the result will be assigned to the variable `sum`. Using a parallel iterator gives an important guarantee that reflects one of the principles of Rust: only one thread is able to access each element of the iterator, which gives complete memory safety and ensures that a data race can never happen.

```
 1 #pragma omp parallel sections
 2 {
 3   #pragma omp section
 4   {
 5     task_a();
 6   }
 7   #pragma omp section
 8   {
 9     task_b();
10   }
11 }
```
Listing 2.4: Parallel tasks in C++

```
 1 rayon::scope(|s|
 2 {
 3     s.spawn(|_|
 4     {
 5         task_a();
 6     });
 7     s.spawn(|_|
 8     {
 9         task_b();
10     });
11 });
```
Listing 2.5: Parallel tasks in Rust

Listings 2.4 and 2.5 shows how two tasks can be executed in parallel in C++ and Rust. The syntax is more similar when executing parallel tasks than in the summation example. In line 1 in Listing 2.4, a compiler directive is used to specify that the following block of code contains OpenMP sections that should be executed in parallel. Then two sections are created in lines 3 and 7, where `task_a()` and `task_b()` are executed in parallel. Each section is executed by a single thread, and each thread can only execute one section.

In line 1 in Listing 2.5, a Rayon scope is created, similar to the `sections` block in Listing 2.4. Inside the scope, two threads are spawned in lines 3 and 7 and are given one task each, also similar to how the `section` blocks are executed in Listing 2.4. The scope enables the execution of parallel tasks, but it also enforces the rules of the ownership model. The main thread will not exit the scope until all threads spawned in the scope are done executing their tasks. This ensures both that the threads will not try to access invalid pointers during execution and that all memory allocated in the scope can safely be freed when the scope is exited.

An unfortunate consequence of the strict rules of the ownership model is that the compiler will not compile a program if one of the rules is broken, even in cases where the developer can guarantee memory safety. The strict rule set may lead to extensive use of atomic variables and mutexes to guarantee safe mutable access to shared resources, which in some cases can result in a lack of performance compared to C++, where access to shared resources is unrestricted.

2.5.3 Synchronisation Mechanisms in Rust

In Rust, data synchronisation is a crucial aspect of multi-threaded programming that ensures the consistency and safety of data across threads. Several synchronisation mechanisms are provided by Rust's standard library. In our implementation, three such techniques are used: `Mutex`, `RwLock`, and `Atomic` variables.

```
    let data = Mutex::new(5);
    {
        *data.lock().unwrap() += 1;
    }
```

Listing 2.6: Example of Mutex in Rust

The `Mutex` type in Rust provides mutual exclusion to a data structure, allowing only one thread to access some data at any given time. To access the data, a thread must first acquire a lock on the `Mutex`. When a thread t has acquired the lock, other threads have to wait to access the data until t releases the lock. Listing 2.6 shows an example of how `Mutex` can protect data. The lock is automatically released when the thread exits the scope that the lock was acquired.

RwLock (Read/Write Lock) is a type that provides a mechanism for read/write locking. It is similar to the Mutex, but is less strict. The RwLock allow multiple readers or one writer to access the data at the same time. The implementation of RwLock is more complex than that of Mutex, but it can lead to better performance when there are many read accesses and few write accesses. Listing 2.7 shows an example of how threads can acquire read and write access to a variable protected by an RwLock. Same as for the Mutex, the lock acquired from a RwLock by a thread is automatically released when the thread exits the scope it was acquired.

```rust
    let data = RwLock::new(5);
    {
        *data.write().unwrap() += 1;
    }
    {
        let r = data.read().unwrap();
    }
```

Listing 2.7: Example of RwLock in Rust

Atomic variables are a type of variable that can be safely accessed and modified by multiple threads concurrently. They are a foundational building block for more complex synchronisation primitives. Listing 2.8 shows how an AtomicUsize can be initialised and atomically incremented.

```rust
    let data = AtomicUsize::new(5);
    data.fetch_add(1, Ordering::SeqCst);
```

Listing 2.8: Example of AtomicUsize in Rust

The specific use case of these synchronisation techniques depends on the implementation of the algorithm and how threads access them, but they are all crucial for developing safe and efficient concurrent programs in Rust.

2.5.4 Summary

Rust's ecosystem includes several powerful yet simple libraries for concurrent programming. The robust low-level concurrency features such as threads, mutexes, channels, and async/await combined with high-level concurrency libraries like Rayon, and guaranteed memory safety from the ownership model, make Rust a safer and more modern alternative to C++ for concurrent programming, ensuring that the resulting code is both comparable performant and significantly more secure. Comparing the performance of the two programming languages is out of the scope of this thesis, and we chose Rust as an interesting alternative to the obvious choice of C++ for high-performance computing.

Chapter 3

Community Detection Algorithms

As mentioned in Section 2.2.1, several algorithms have been developed to detect communities in static graphs, each with its strengths and weaknesses, and one of the most used algorithms is the Louvain algorithm.

The Louvain algorithm [4] is a fast and scalable algorithm that aims to optimise the modularity of a partitioning of the graph into communities by iteratively moving vertices between communities. While the Louvain algorithm has been shown to perform well on many types of graphs, it can sometimes produce sub-optimal results on certain types of graphs with specific structural properties.

The Leiden algorithm [29] is an improved version of the Louvain algorithm. It addresses some limitations, such as the tendency to merge small communities, create disconnected communities and the sensitivity to the order in which vertices are processed. Overall, the Leiden algorithm has been shown to outperform the Louvain algorithm on several benchmarks and real-world networks [29, pp. 9-11], and it has become a popular choice for community detection in many fields.

3.1 The Louvain Algorithm

The Louvain algorithm is a widely used community detection heuristic in network science, introduced by Blondel et al. in 2008 [4]. It is known for its ability to detect communities in large graphs efficiently and has several advantages over other community detection algorithms. It is computationally efficient and can handle very large graphs, is reasonably easy to implement and does not require prior knowledge about the graph's layout.

Algorithm 1 The Louvain algorithm

```
1: function LOUVAIN(Graph $G$, Partion $P$)
2:     do
3:         $\mathcal{P} \leftarrow$ MOVEVERTICES($G, \mathcal{P}$)          ▷ Move vertices between communities
4:         done $\leftarrow |\mathcal{P}| = |v(G)|$          ▷ Terminate when each community consists of only one vertex
5:         if not done then
6:             $G \leftarrow$ AGGREGATEGRAPH($G, \mathcal{P}_{refined}$)          ▷ Create aggregate graph based on partition $\mathcal{P}$
7:             $\mathcal{P} \leftarrow$ SINGLETON($G$)          ▷ Assign each vertex in aggregate graph to its own community
8:         end if
9:     while not done
10:     return flat*($\mathcal{P}$)
11: end function
```

Algorithm 1 shows the outline of the Louvain algorithm. The algorithm repeatedly improves the partition $\mathcal{P}$ and aggregates the graph until each community consists of only one vertex. Algorithms 2 and 3 explains in depth how the moving and aggregation works.

Algorithm 2 Local moving phase of the Louvain algorithm

```
1: function MOVEVERTICES(Graph $G$, Partition $\mathcal{P}$)
2:     do
3:         $Q_{old} = Q(\mathcal{P})$
4:         for $v \in V(G)$ do          ▷ Visit vertices (in random order)
5:             $C' \leftarrow \arg\max_{C \in \mathcal{P} \cup \emptyset} \Delta Q_{\mathcal{P}}(v \mapsto C)$          ▷ Determine the best cluster for vertex $v$
6:             if $\Delta Q_{\mathcal{P}}(v \mapsto C') > 0$ then          ▷ Perform only strict positive vertex movements
7:                 $v \mapsto C'$          ▷ Move vertex $v$ to cluster $C'$
8:             end if
9:         end for
10:     while $Q(\mathcal{P}) > Q_{old}$          ▷ Continue as long as the modularity increases
11:     return $\mathcal{P}$
12: end function
```

Algorithm 2 shows the local moving phase of the Louvain algorithm. The algorithm works by iteratively moving vertices between communities to maximise the modularity of the graph partition. The outermost do-while loop in line 2 runs as long as modularity increases. In each iteration, the algorithm determines the best move for each vertex (line 5). This could be to one of the neighbouring communities or to a singleton community. If there is a positive gain in modularity, the vertex is moved in line 7, otherwise it remains in its current community. This process is repeated until no further increase in modularity is possible, which is checked in line 10. The moving phase of the algorithm is depicted from step **a)** to **b)** in Figure 3.1

Algorithm 3 Aggregation phase of the Louvain algorithm

```
1: function AGGREGATEGRAPH(Graph $G$, Partition $\mathcal{P}$)
2:     $V \leftarrow \mathcal{P}$          ▷ Communities become vertices in aggregate graph
3:     $E \leftarrow \{(C, D) \mid (u, v) \in E(G), u \in C \in \mathcal{P}, v \in D \in \mathcal{P}\}$ ▷ Edges between vertices in community $C$ and $D$ are merged
4:     return GRAPH($V$, $E$)
5: end function
```

When all vertices are stable, i.e. moving a vertex will not increase modularity, the graph is aggregated into a new graph with a coarser level of granularity. The aggregation is based on the partitioning of the vertices from the previous phase, shown in Algorithm 3. On line 2 the set of vertices in the new graph is set to be equal to the set of communities in $\mathcal{P}$, and on line 3, the set of edges in the new graph is created such that if there is an edge between vertices u and v in the old graph, there should be an edge between the vertices representing their respective communities C and D. The aggregation phase is depicted from step **b)** to **c)** in Figure 3.1 [29, p. 2].

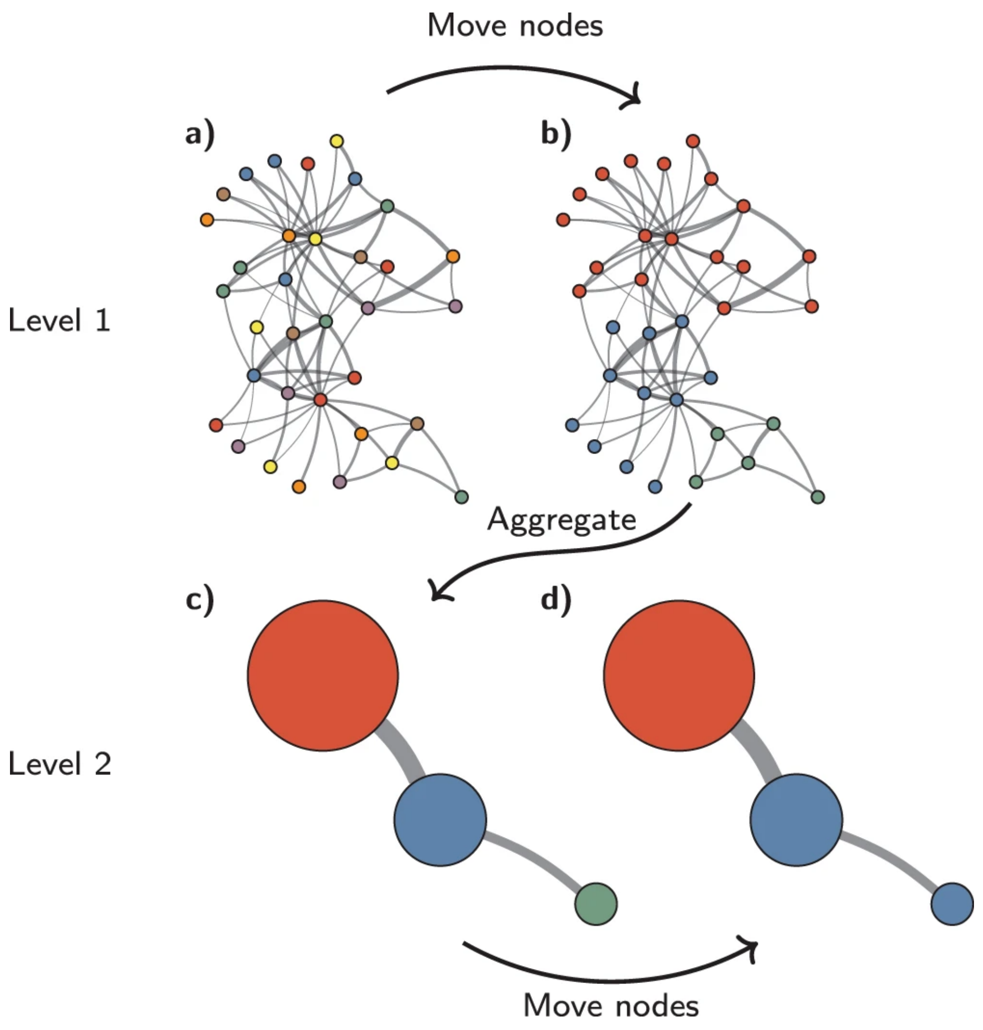


Figure 3.1: Steps of the Louvain algorithm

Despite the efficiency and simplicity of the Louvain algorithm, it does have its limitations [29, pp. 2-4]. One major weakness is the resolution limit of the modularity, which refers to the minimum size of the communities that can be detected by the algorithm. At each level of the algorithm, it tries to move vertices to other communities to increase modularity, and when the size of a community C is smaller than a certain threshold, the algorithm is more likely to merge C into another community because doing so may improve modularity more than keeping C on its own. The Louvain algorithm also has a tendency to produce badly connected or unconnected communities, which can result in sub-optimal modularity. Another weakness is that it tends to produce different community structures depending on the initial conditions, leading to instability in the results. Also, the type and structure of the graph may greatly affect the algorithm's ability to optimise the modularity.

3.2 The Leiden Algorithm

The Leiden algorithm is a community detection heuristic that was first introduced by Traag et al. in 2019 [29]. It is a refinement of the Louvain algorithm, and like the Louvain algorithm, the Leiden algorithm is a hierarchical clustering algorithm that works by iteratively moving vertices between communities to increase modularity. However, unlike the Louvain algorithm, the Leiden algorithm uses a refinement strategy to optimise the modularity even further, which takes into account both the internal connectivity of communities and their degree of separation from each other. This improvement strategy allows the Leiden algorithm to more effectively identify communities of vertices that are densely connected to each other and well-separated from the rest of the graph.

Algorithm 4 The Leiden algorithm

```
1: function LEIDEN(Graph $G$, Partion $P$)
2:     do
3:         $\mathcal{P} \leftarrow$ MOVEVERTICESFAST($G, \mathcal{P}$)        ▷ Move vertices between communities
4:         done $\leftarrow |\mathcal{P}| = |v(G)|$        ▷ Terminate when each community consists of only one vertex
5:         if not done then
6:             $\mathcal{P}_{\text{refined}} \leftarrow$ REFINEPARTITION($G, \mathcal{P}$)        ▷ Refine partition $\mathcal{P}$
7:             $G \leftarrow$ AGGREGATEGRAPH($G, \mathcal{P}_{refined}$)        ▷ Create aggregate graph based on refined partition $\mathcal{P}_{refined}$
8:             $\mathcal{P} \leftarrow \{\{v \mid v \subseteq C, v \in V(G)\} \mid C \in \mathcal{P}\}$        ▷ But maintain partition $\mathcal{P}$
9:         end if
10:    while not done
11:    return flat$^*(\mathcal{P})$
12: end function
```

Algorithm 4 shows the outline of the Leiden algorithm. It looks similar to the Louvain algorithm, except for the refinement step in line 6. In addition to the refinement step, the partition $\mathcal{P}$ after the aggregation of G is a little different. Algorithms 5 and 6 show the improved local moving and the refinement step of the Leiden algorithm. The AGGREGATEGRAPH function used in the Leiden algorithm is the same as in the Louvain algorithm, explained in Algorithm 3.

Algorithm 5 Local moving phase of the Leiden algorithm

```
1: function MoveVerticesFast(Graph $G$, Partition $\mathcal{P}$)
2:     $Q \leftarrow$ Queue($V(G)$)                    ▷ Add all vertices to queue (in random order)
3:     do
4:         $v \leftarrow Q$.remove()                      ▷ Pop next vertex from queue
5:         $C' \leftarrow \arg\max_{C \in \mathcal{P} \cup \emptyset} \Delta H_{\mathcal{P}}(v \mapsto C)$     ▷ Determine the best community for vertex $v$
6:         if $\Delta H_{\mathcal{P}}(v \mapsto C') > 0$ then      ▷ Perform only strict positive vertex movements
7:             $v \mapsto C'$                            ▷ Move vertex $v$ to cluster $C'$
8:             $N \leftarrow \{u \mid \{u, v\} \in E(G), u \notin C'\}$   ▷ Identify neighbours of vertex $v$ that are not in community $C'$
9:             $Q$.add($N - Q$)                         ▷ Add non-queued neighbours to queue
10:        end if
11:    while $Q \neq \emptyset$                      ▷ Continue until there are no more vertices to process
12:    return $\mathcal{P}$
13: end function
```

Algorithm 5 shows the local moving phase of the Leiden algorithm. It starts in the same way as the Louvain algorithm, with each vertex being a community on its own. Then, the algorithm iteratively moves vertices between communities to improve modularity. At each iteration, the Leiden algorithm considers moving each vertex to its neighbouring communities and calculates the change in modularity that would result from the move. The algorithm then selects the move that results in the greatest increase in modularity and makes that move.

At this point, the Leiden algorithm differs from the Louvain algorithm. In the Louvain algorithm, if at least one vertex is moved to a new cluster, the local moving phase will repeat and iterate through all the vertices of the graph. The local moving phase in the Leiden algorithm processes a queue of vertices initiated by all vertices in the graph and proceeds until the queue is empty. When a vertex v is moved to a new community C', all the neighbours of v that belong to a different community than C' (line 8) and are not already in the queue will be added to the queue, as shown on line 9. This results in significantly fewer vertices being processed; hence, lower running time. The process is repeated until no further moves can be made that result in an increase in modularity. Figure 3.2 step **a)** to **b)** shows how the local moving phase is initiated with a singleton partition and results in three communities of different sizes.

The largest improvement from Louvain to Leiden is the refinement phase, depicted in Algorithm 6. Whereas the Louvain algorithm jumps from the local moving phase straight to the aggregation of the graph, the Leiden algorithm tries to refine the partition before aggregating the graph. In the refinement phase, the primary objective of the Leiden algorithm is to optimise and enhance the quality of the partition discovered during the local moving phase.

Algorithm 6 Refinement phase of the Leiden algorithm

```
1: function RefinePartition(Graph $G$, Partition $\mathcal{P}$)
2:     $\mathcal{P}_{\text{refined}} \leftarrow$ SingletonPartition($G$)          ▷ Assign each vertex to its own community
3:     for $C \in \mathcal{P}$ do                                               ▷ Visit communities
4:         $\mathcal{P}_{\text{refined}} \leftarrow$ RefineCommunity($G, \mathcal{P}_{\text{refined}}, C$)   ▷ Refine community $C$
5:     end for
6:     return $\mathcal{P}_{\text{refined}}$
7: end function

8: function RefineCommunity(Graph $G$, Partition $\mathcal{P}$, Subset $S$)
9:     $R \leftarrow \{v \mid v \in S, E(v, S - v) \geq \gamma \|v\| \cdot (\|S\| - \|v\|)\}$   ▷ Find all well-connected vertices
10:    for $v \in R$ do                                                ▷ Visit vertices (in random order)
11:        if $|\mathcal{P}(v)| = 1$ then                       ▷ Consider only vertices in singleton communities
12:            $\mathcal{T} \leftarrow \{C \mid C \in \mathcal{P}, C \subseteq S, E(C, S - C) \geq \gamma \|C\| \cdot (\|S\| - \|C\|)\}$   ▷ Find all well-connected communities
13:            $\Pr(C' = C) \sim \begin{cases} \exp(\frac{1}{\theta}\Delta Q_{\mathcal{P}}(v \mapsto C)) & \text{if } Q_{\mathcal{P}}(v \mapsto C) \geq 0 \\ 0 & \text{otherwise} \end{cases}$ for $C \in \mathcal{T}$   ▷ Choose random cluster $C'$
14:            $v \mapsto C'$                                          ▷ Move vertex $v$ to cluster $C'$
15:        end if
16:    end for
17:    return $\mathcal{P}$
18: end function
```

The refinement process starts by initialising a partition, where each vertex is assigned to its own community (line 2) and then iteratively refines each community in the partition produced by the local moving phase. For each community S, a set R of vertices that are well-connected within S is computed on line 9. This set is determined by comparing the edge connections of each vertex within S to a threshold value that depends on the chosen resolution parameter and the total edge weight within S. The algorithm then iterates through each vertex in the set of well-connected vertices in random order, considering only vertices that are in a singleton community.

On line 12, the algorithm computes a set $\mathcal{T}$ of all well-connected neighbour communities within community S. This ensures that only communities with strong connections relative to their size are considered. Once the set of well-connected neighbour communities is identified, the algorithm computes the probability of moving a vertex v to a different community within this set. This probability is based on the change in the modularity caused by moving v to a different community and is determined using an exponential function of the change in modularity. The degree of randomness in the selection of a community is determined by a parameter $\theta > 0$. The algorithm only considers moving v to communities that result in a non-negative modularity change. After computing the probabilities, the algorithm randomly selects a community from the set of well-connected communities based on the calculated probabilities and moves v to the chosen community.

After the refinement phase is concluded, communities in $\mathcal{P}$ often will have been split into multiple communities in $\mathcal{P}_{refined}$, but not always. As a result of Leiden detecting communities within communities, it tends to produce more communities than Louvain. The refinement phase is critical in improving the overall performance of the Leiden algorithm and ensuring that the community partition is more accurate and representative of the graph structure compared to other modularity optimisation algorithms, such as the Louvain algorithm [29, pp. 7-11].

The refinement phase is depicted in Figure 3.2 [29, p. 4] from state **b)** to **c)** where it detects two new communities within the red and green communities, resulting in a partition consisting of five communities being aggregated into a graph, whereas Louvain only aggregated three communities. The aggregate phase of the Leiden algorithm is identical to that of the Louvain algorithm, portrayed in Algorithm 3.

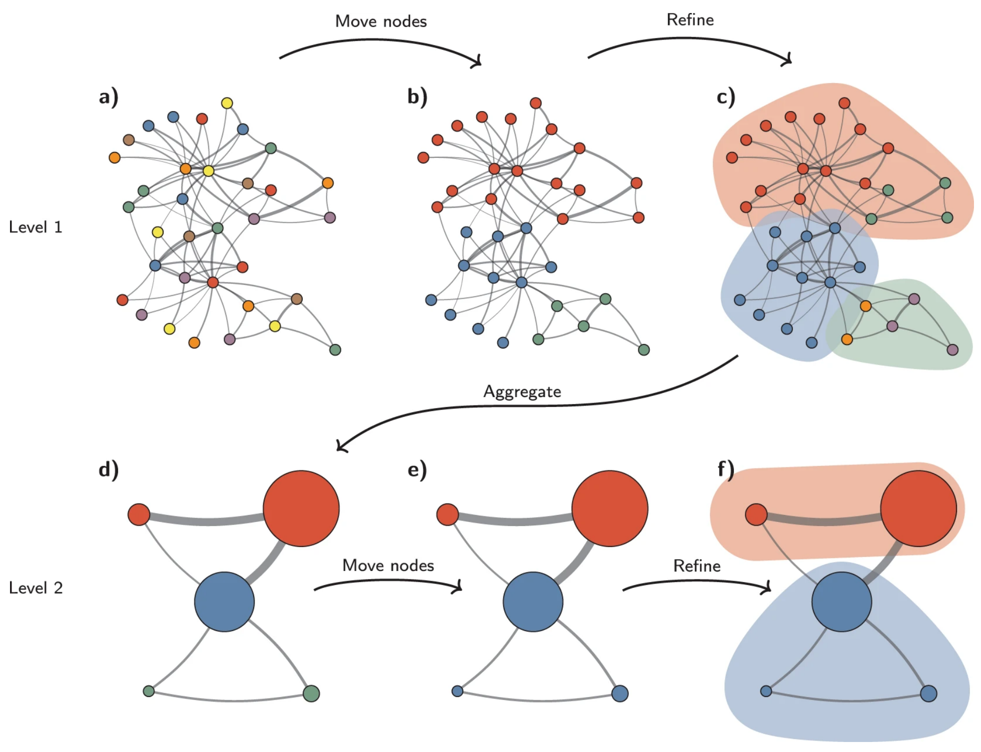


Figure 3.2: Steps of the Leiden algorithm

The Leiden algorithm has been shown to be highly effective in identifying communities in a wide range of complex real-world networks, including social networks, biological networks, and technological networks [29, pp. 10-11]. It has also been shown to outperform other state-of-the-art community detection algorithms in terms of both accuracy and speed. Even though the Leiden algorithm is one of the more efficient algorithms for community detection in static graphs, it falls behind when trying to detect communities in incremental graphs. If the Leiden algorithm were to be used to cluster an incremental graph, the whole graph would have to be reclustered every time new data arrived, which would quickly lead to an unviable execution time. Hence, we need an algorithm designed specifically for incremental graphs.

3.3 The NCLiC Algorithm

The Neighbourhood-to-Community Link Counting (NCLiC) algorithm is an algorithm designed for community detection in incremental graphs, introduced by Tumanis in 2021 [31, pp. 79-85]. The algorithm works by continuously processing chunks of edges, represented by two vertices, which may be either two vertices already processed, one old and one new vertex, or two new vertices. First, the vertices in the arriving chunks are clustered using the Leiden algorithm. The partition is then merged with the communities of previously processed vertices using a refinement step. The refinement phase is based on the assumption that if a majority of the neighbours of a vertex v belong to a community, there is a high probability that v belongs to the same community. The algorithm also assumes that the Leiden algorithm produces local partitions that are close to optimal. The aim of the refinement phase is to merge the partitions with the lowest modularity loss possible.

3.3.1 Data Structures

The NCLiC algorithm uses three data structures to store data of all previously processed vertices:

- $\boldsymbol{G}$: A dynamic graph consisting of the vertices and edges from processed chunks.
- $\mathcal{P}$: A dynamic array to keep track of the current community of vertices in $V(G)$.
- $\boldsymbol{NCC}$: A dynamic array consisting of $|V(G)|$ hash maps. The hash map at index i keeps track of the communities of the vertices adjacent to the vertex with id i.

Data Structure 1 Dynamic graph data structure

```
Graph {
    n_vertices: Integer representing the number of vertices in the graph
    n_edges: Integer representing the number of edges in the graph
    adj_lists: Dynamic array of linked lists representing the adjacency lists with edge weights for each vertex
    vertex_weights: Dynamic array of sums of edge weights incident to each vertex
    label_to_id: Associative array mapping vertex labels to ids
    id_to_label: Dynamic array mapping vertex ids to labels
}
```

The graph structure shown in Data Structure 1 uses mainly four data structures to store information about the graph. The variables `n_vertices` and `n_edges` are integers that store the number of vertices and edges in the graph. Variable `adj_lists` is a dynamic array with `n_vertices` elements, where index i of the array contains a pointer to a linked list containing the neighbours and edge weights of the vertex with id i. If the edge weights are not given as part of the chunk, the weight is set to 1. Variable `vertex_weights` is a dynamic array with `n_vertices` elements, where index i of the array contains an integer representing the weight of the vertex with id i. The weight of a vertex v is the sum of the edge weights of the edges incident to v.

To reduce the amount of memory needed and to exploit the advantages of processing data stored in contiguous memory sections [19], all the vertices are assigned a zero-indexed id. Thus, two more data structures are needed to store mappings from label to id and vice-versa. Variable `label_to_id` is a hash map mapping the original label of the vertices to ids and variable `id_to_label` is a dynamic array of size $|V(G)|$ that maps the assigned ids to the original labels. Another benefit of relabelling the new vertices when they are merged into G is that the partition $\mathcal{P}$ and the neighbour community count NCC can be stored in dynamic arrays of size $|V(G)|$, avoiding unnecessary allocation of memory.

3.3.2 Implementation

Algorithm 7 shows the outline of the NCLiC algorithm. First, the algorithm initialises three data structures to store information about already processed chunks: an empty graph G, an empty partition $\mathcal{P}$ and an empty array NCC. Chunks are then processed iteratively and for each iteration, four steps are applied:

1. Build a graph G_i from the edges in chunk S_i and a singleton partition $\mathcal{P}_i$ of the vertices in G_i, shown on lines 6 through 9
2. Pre-cluster G_i using the Leiden algorithm and merge the communities of the new vertices into $\mathcal{P}$, shown on line 10
3. Merge the vertices and edges of G_i into G, shown in lines 11 and 12
4. Refine the new partition based on the neighbour community count, shown on line 13

Algorithm 7 NCLiC algorithm

```
1: function NCLiC(Data stream $S$)
2:     $G \leftarrow$ Graph()                                   ▷ Initialise empty graph
3:     $\mathcal{P} \leftarrow$ empty array                          ▷ Initialise empty partition
4:     $NCC \leftarrow$ empty array        ▷ Initialise empty array for neighbour community count
5:     while $S$ has next do
6:         Read $S_i$                                  ▷ Read chunk of edges from $S$
7:         $V(G_i) \leftarrow \{v \mid \{u,v\} \in S_i$ or $\{v,u\} \in S_i\}$
8:         $E(G_i) \leftarrow \{e \mid e \in S_i\}$                    ▷ Initialise graph from chunk
9:         $\mathcal{P}_i \leftarrow$ SingletonPartition($G_i$)      ▷ Initialise singleton partition for chunk
10:        $\mathcal{P}(V(G_i) \setminus V(G)) \leftarrow$ Leiden($G_i, \mathcal{P}_i$)   ▷ Pre-cluster the chunk and store communities of new vertices
11:        $V(G) \leftarrow V(G) \cup V(G_i)$                  ▷ Merge chunk into the main graph
12:        $E(G) \leftarrow E(G) \cup E(G_i)$
13:        RefineCommunities($\mathcal{P}, G, G_i, NCC$)          ▷ Refine the pre-clustered chunk
14:    end while
15: end function
```

3.3.3 Refinement

Algorithm 8 shows the refinement phase of the NCLiC algorithm. The algorithm iterates through each vertex v in the new graph G_i. If v has not been part of an earlier chunk, it is assigned an empty neighbour community count on line 4. Then, on line 7, the NCC of v will be updated with the communities of the neighbours of v in G_i. On line 9, the algorithm finds the community most common among the neighbours of v. In case of a tie, a community is chosen randomly between the most common communities. If the chosen community C differs from the current one, two things will happen. First, vertex v is moved to C. Second, the algorithm calculates the probability of whether the neighbour community count of the neighbours of v in G should be updated. This is based on the difference between the average degree in G and the degree of v, which means that vertices with a low degree compared to the average degree have a higher chance of being updated than those with a high degree. The reason that the probability decreases with increasing degree is based on the assumption that a vertex with several neighbours has a higher probability of being in the "best" community compared to vertices with few neighbours.

Algorithm 8 Refinement phase of the NCLiC algorithm

```
1: function RefineCommunities(Partition $\mathcal{P}$, Graph $G$, Graph $G_i$, HashMap $NCC$)
2:     for $v \in V(G_i)$ do                                   ▷ Iterate through vertices in $G_i$
3:         if $v \notin V(G)$ then
4:             $NCC[v] \leftarrow$ empty hash map              ▷ Initialise neighbour community count for new vertex $v$
5:         end if
6:         for $w \in N_{G_i}(v)$ do
7:             $NCC[v][\mathcal{P}(w)] \leftarrow +1$          ▷ Increment count of all neighbour communities of $v$
8:         end for
9:         $C_{\text{new}} \leftarrow \arg\max(NCC[v])$        ▷ Find the community $C_{\text{new}}$ that most neighbours of $v$ are assigned to
10:        if $\mathcal{P}(v) \neq C_{new}$ then
11:            $C_{\text{old}} \leftarrow \mathcal{P}(v)$
12:            $v \mapsto C_{new}$                             ▷ Move $v$ to $C_{\text{new}}$
13:            $x \leftarrow \texttt{avgdegree} - \texttt{degree}(v)$
14:            $p \leftarrow \left(\frac{x}{\sqrt{1+x^2}} + 1\right)/2$   ▷ Calculate the probability that the NCC of neighbours should be updated
15:            if $\texttt{random} < p$ then
16:                for $w \in N_G(v)$ do                       ▷ Update $NCC$ for all neighbours of $v$
17:                    $NCC[w][C_{\text{old}}] \leftarrow -1$  ▷ Decrease count of old community
18:                    $NCC[w][C_{\text{new}}] \leftarrow +1$  ▷ Increase count of new community
19:                end for
20:            end if
21:        end if
22:    end for
23: end function
```

When a chunk is processed, the Leiden algorithm produces a community partition with close-to-optimal modularity locally. However, this might not be the optimal choice of community for all vertices when merged into graph G. The refinement phase ensures that the final community partition in the whole graph G is more accurate and that the modularity loss is minimised.

3.3.4 NCLiC Steps

The example in Figure 3.3 [31, p. 80] shows how the NCLiC algorithm processes one chunk. The large, grey boxes in the steps represent the graph G_i built from the incoming chunk S_i. The small grey boxes represent graph G, consisting of previously processed vertices and edges. Vertices 5 and 6 have been part of an earlier chunk and are already assigned communities. However, edges $\{5,7\}$, $\{5,8\}$ and $\{6,8\}$ are part of chunk S_i, hence, the vertices are processed in this iteration as well.

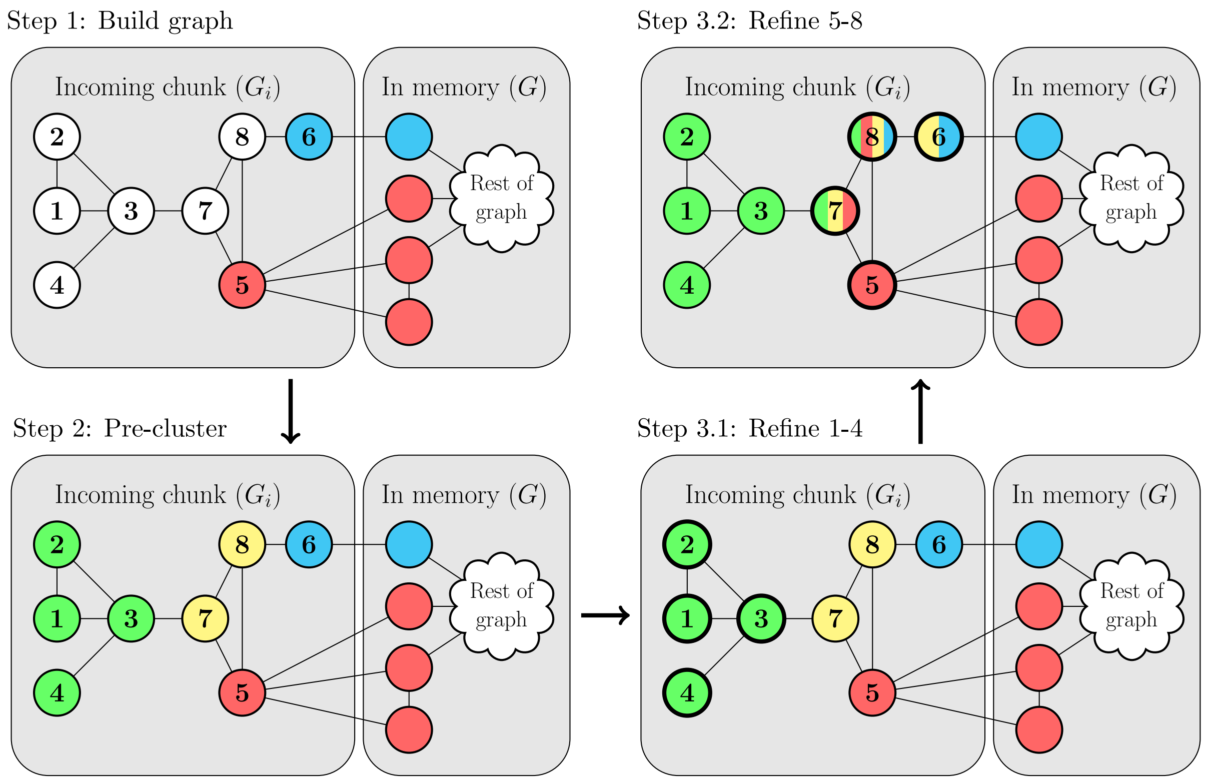


Figure 3.3: Steps of the NCLiC algorithm

Step 1: Build graph Chunk S_i is read from the data stream followed by the construction of the graph G_i. Note that vertices 5 and 6 have been part of an earlier chunk and are already placed in the red and the blue community, respectively.

Step 2: Pre-cluster The Leiden algorithm is used to partition the vertices into communities that yield the highest modularity score. Vertices 5 and 6 are already assigned communities, and the result of the Leiden algorithm does therefore not affect them. The rest of the vertices are partitioned into two communities, indicated by green and yellow.

Step 3.1: Refine vertices 1 through 4 The refinement phase iteratively processes vertices 1 through 4. However, because the majority of their neighbours are placed in the green community, none of the vertices will be moved to another community.

Step 3.2: Refine vertices 5 through 8 The refinement phase iteratively processes vertices 5 through 8.

- Vertex 5 has three neighbours in graph G placed in the red community and two neighbours placed in the yellow community; hence, vertex 5 will remain in the red community.
- Vertex 6 has two neighbours, vertex 8 in the yellow community and one vertex in graph G in the blue community. There is a tie, and vertex 6 will randomly choose between remaining in the blue community or moving to the yellow community.
- Vertex 7 has three neighbours assigned to different communities, green, yellow and red. Thus it will be moved to either the green, the red or remain in the yellow community. This is again chosen randomly.
- The results of the random choices for vertices 6 and 7 decides what happens to vertex 8. The possible combinations of neighbouring communities are:
  - Red (5), green/yellow (7), blue (6) $\rightarrow$ vertex 8 is randomly placed in one of the four communities.
  - Red (5), green (7), yellow (6) $\rightarrow$ vertex 8 is randomly placed in one of the three communities.
  - Red (5), yellow (7), yellow (6) $\rightarrow$ vertex 8 has two neighbours in the yellow community and will remain there.
  - Red (5), red (7), yellow/blue (6) $\rightarrow$ vertex 8 has two neighbours in the red community and will be moved to this.

The example in Figure 3.3 shows both the strengths and the weaknesses of the NCLiC algorithm. Vertices with few previously processed neighbours are likely to be moved around several times, for instance, vertex 6, which may be the only vertex placed in the yellow community after the refinement step is done. This will, however, be stabilised when several chunks have been processed, as seen for vertex 5, which has enough neighbours in the red community in G to remain there, regardless of the local partition of G_i. It is important to note that because of the neighbour community count, NCC, the algorithm can decide that vertex 5 is stable without actually iterating through its neighbours in G.

3.3.5 NCLiC Example

Figures 3.5 through 3.8 [31, pp. 82-84] depict how the NCLiC algorithm works when processing the graph in Figure 3.4 represented as four chunks of edges.

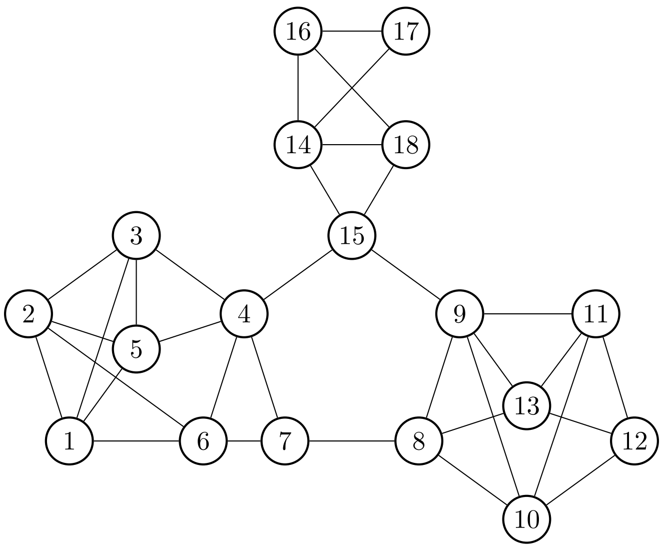


Figure 3.4: A graph to be clustered by the NCLiC algorithm

The graph in Figure 3.4 shows a clear community structure consisting of three clusters.

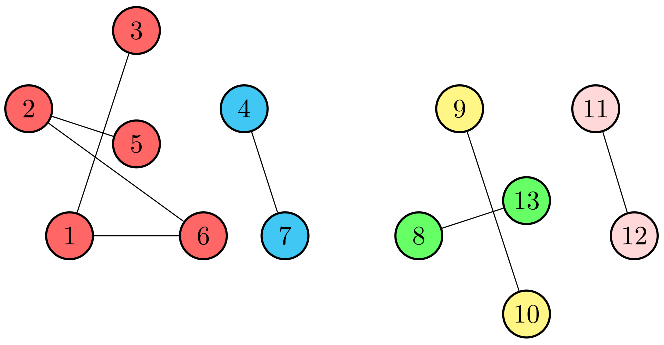


Figure 3.5: Graph G after the initial chunk of edges is processed

Figure 3.5 shows the first chunk, which consists of the following edges:

$$\{\{1,3\},\{1,6\},\{2,5\},\{2,6\},\{4,7\},\{8,13\},\{9,10\},\{11,12\}\}$$

After running the Leiden algorithm on the chunk, the vertices are partitioned into five communities. When processing the initial chunk, the algorithm skips the refinement phase.

This is because running the refinement step would, at best, result in the same partition as the Leiden algorithm did.

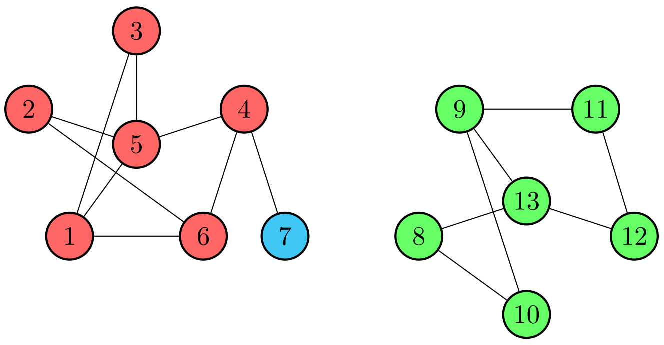


Figure 3.6: Graph G after merging of the second chunk of edges

Figure 3.6 shows the result given by the NCLiC algorithm after the second chunk has been processed. This chunk consists of the following edges:

$$\{\{1, 5\}, \{3, 5\}, \{4, 5\}, \{4, 6\}, \{8, 10\}, \{9, 11\}, \{9, 13\}, \{12, 13\}\}$$

Vertices $1, 2, 3, 5$ and 6 were already part of the red community and remain there. Vertex 4 has gotten two new neighbours in this chunk, 5 and 6, which are both assigned to the red community; hence, the refinement phase moves vertex 4 to the red community. Vertex 7 is not part of any edges in the chunk and will not be processed in the refinement phase, even though it has a neighbour in the red community. Vertices 8 through 13 will all be moved into the same community as a result of running the Leiden algorithm and the refinement phase.

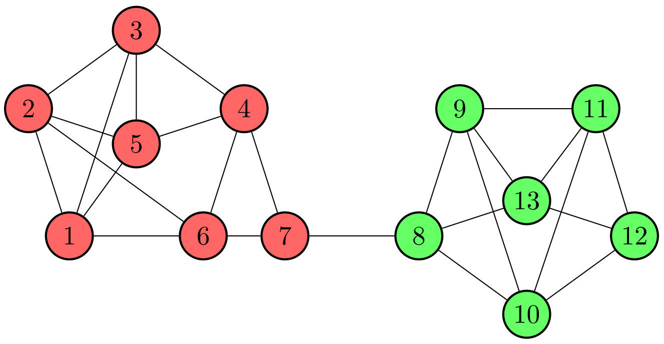


Figure 3.7: Graph G after merging of the third chunk of edges

Figure 3.7 shows the result given by the NCLiC algorithm after the third chunk has been processed. This chunk consists of the following edges:

$$\{\{1, 2\}, \{2, 3\}, \{3, 4\}, \{6, 7\}, \{7, 8\}, \{8, 9\}, \{10, 11\}, \{10, 12\}, \{11, 13\}\}$$

Even though there are several new edges, only one vertex changes community. Vertex 7 has two neighbours in the red community and one neighbour in the green community, and the refinement phase moves the vertex to the community that most of its neighbours are assigned to.

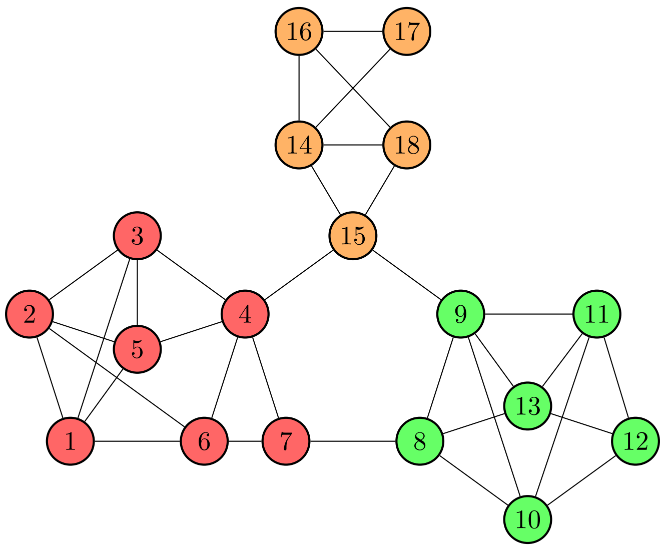


Figure 3.8: Graph G after merging of the last chunk of edges

Figure 3.8 shows the result given by the NCLiC algorithm after the last chunk has been processed. This chunk consists of the following edges:

$$\{\{4,15\},\{9,15\},\{14,15\},\{14,16\},\{14,17\},\{14,18\},\{15,18\},\{16,17\},\{16,18\}\}$$

The Leiden algorithm assigns all the new vertices 14 through 18 to one community: orange. The only vertex with neighbours in another community is vertex 15, but with two neighbours in the orange community, the refinement phase does not affect it. When all chunks are processed, the vertices are partitioned into three communities that display the clear community structure of the graph.

The NCLiC algorithm has been shown to be highly effective in clustering incremental graphs [31, p. 102]. It far outperforms offline algorithms while retaining a significant fraction of the quality of the communities. However, as the amount of data in real-world networks increases rapidly, there is a need for even further development of this algorithm to handle the scale and complexity of these networks, raising the question for this thesis: **How can the Neighbourhood-to-Community Link Counting algorithm be parallelised?**

Chapter 4

Related Work

4.1 Parallelised Community Detection

To the best of our knowledge, there exists little research on the parallelisation of algorithms designed for community detection in streaming and incremental graphs. There do, however, exist several approaches to parallelise community detection in static graphs, using different community detection algorithms with different parallelisation techniques. In 2018 Zeng et al. introduced an MPI implementation of the Infomap algorithm [35], obtaining speedups up to a factor of 6 compared to the sequential implementation. Naim et al. introduced a GPU implementation of the Louvain algorithm in 2017 [23], obtaining speedups up to a factor of 270 compared to the sequential algorithm.

Even though the Leiden algorithm is a relatively recent algorithm, parallelisation of the algorithm has been explored to some extent. A parallel implementation of the local moving phase of the algorithm was introduced by Verweij in 2019 [32]. The algorithm obtained a speed-up of up to 1.65, depending on the size and density of the graph being clustered.

A bachelor thesis written by Nguyen in 2021 [27] introduced a parallel implementation of the Leiden algorithm using a shared memory approach, where both the local moving phase and the refinement phase were parallelised. The parallel algorithm achieved up to 28 times speed up for the local moving phase and up to 15 times for the refinement phase, and preserving virtually the same modularity as the sequential implementation.

Chapter 5

Parallel NCLiC

In this chapter, we present the details of our novel implementation of the parallel NCLiC algorithm. The algorithm repeatedly applies four steps: graph building, pre-clustering, graph merging and refinement, shown in Section 3.3.4. We consider if and how these can be parallelised and which precautions must be taken, and we present the parallel implementation of the steps. The advantages and limitations of the implementations are also discussed.

5.1 Building the Graph

The first step when a new chunk arrives is building a graph G_i from the edges of the chunk. The graph building is done sequentially in the parallel implementation of the algorithm. If the graph was to be built in parallel, the graph structure would have to be implemented with several synchronisation mechanisms, such as mutexes and atomic variables, to ensure memory safety and eliminate the possibility of race conditions, according to the rules of the ownership model in Rust, explained in Section 2.5.1.

When performing parallel pre-clustering, G_i is only read by multiple threads and never written to, which means that an immutable reference to the graph structure could be shared between threads without further synchronisation mechanisms while still obeying the rules of the ownership model. The pre-clustering of G_i is a significantly more time-consuming operation than building it, thus, running the clustering algorithm on a graph structure containing synchronisation mechanisms would lead to more overhead than building the graph in parallel would speed up the process.

5.2 Pre-clustering

In this section, we cover our implementation of the parallel Leiden algorithm, inspired by the implementation by Nguyen [27]. The Leiden algorithm consists of three steps, explained in Section 3.2. Algorithms 9 through 12 explains the parallel implementation of the fast local moving and refinement phases.

5.2.1 Parallel Local Moving

The pseudocode of the parallel local moving phase is split in three, Algorithms 9 through 11, to improve readability. First, we define the shared and thread-local data structure used in the implementation:

- Global
  - `global_Q`: A shared thread-safe queue of arrays of vertices that should be processed.
  - `in_queue`: A shared array of size $|V(G)|$ where the atomic boolean at index i determines whether the vertex with id i is in the queue or not.
  - `threads_waiting`: A shared atomic integer representing the number of threads currently waiting for work.
- Local
  - `local_Q`: A thread-local queue initialised for each thread consisting of vertices that should be processed. When a thread pops an array of vertices from the global queue, the vertices are pushed to the local queue.
  - `new_queue`: A thread-local array where each thread store vertices that should be reprocessed.

Algorithm 9 Local moving phase of the parallel Leiden algorithm

```
1: function ParallelMoveVerticesFast(Graph $G$, Partition $\mathcal{P}$)
2:     global_Q ← empty queue                    ▷ Initialise thread-safe empty global queue
3:     in_queue ← [true for $v \in V(G)$]          ▷ Set all vertices as in queue
4:     threads_waiting ← 0                        ▷ Initialise atomic integer to keep track of waiting threads
5:     vertices_per_thread ← $|V(G)|$ / num_threads   ▷ Split vertices evenly among threads
6:     - Open parallel region -
7:     first_vertex ← thread_id × vertices_per_thread
8:     last_vertex ← (thread_id + 1) × vertices_per_thread
9:     local_Q ← [ first_vertex, last_vertex )    ▷ Initialise local queue
10:    while local_Q ≠ ∅ do                      ▷ Work as long as the local queue is not empty
11:        Move vertices in the local queue (Algorithm 10)
12:        Get work from the local or global queue or wait (Algorithm 11
13:    end while
14:    - Close parallel region -
15:    return $\mathcal{P}$
16: end function
```

Algorithm 9 shows the outline of the parallel local moving phase. In lines 2 through 5 the global variables are initialised. In line 6, the threads are initialised and are assigned non-overlapping sets of vertices in line 9. As long as a thread t has vertices in the local queue, it will continue working. In line 11, t processes the vertices in its local queue, explained in Algorithm 10 and when all vertices in the local queue have been processed, t proceeds to line 12, where it either gets more vertices to process or wait for another thread to push work to the global queue. This step is explained in Algorithm 11.

Algorithm 10 Working threads in the parallel local moving phase

```
1: new_queue ← empty queue                                ▷ Initialise empty queue to keep track of neighbour vertices
2: for $v \in$ local_Q do                                  ▷ Iterate through vertices in local queue
3:     $C' \leftarrow \arg\max_{C \in \mathcal{P} \cup \emptyset} \Delta H_{\mathcal{P}}(v \mapsto C)$     ▷ Determine the best community for vertex $v$
4:     if $\Delta H_{\mathcal{P}}(v \mapsto C') > 0$ then              ▷ Perform only strict positive vertex movements
5:         $v \mapsto C'$                                      ▷ Move vertex $v$ to cluster $C'$
6:         for $w \in N_G(v)$ do                           ▷ Iterate through neighbours of $v$
7:             if CompareAndSwap($w$) and $P(w) \neq C'$ then
8:                 new_queue.push($w$)                     ▷ Add neighbour to the new queue
9:                 if new_queue.size() = 1000 then
10:                    global_Q.push(new_queue)            ▷ Push new queue to global queue
11:                    WakeAllThreads()                    ▷ Notify waiting threads that there is available work
12:                    new_queue ← empty queue             ▷ Empty new queue
13:                end if
14:            end if
15:        end for
16:        in_queue[$v$] ← false                          ▷ Set vertex $v$ as not in queue
17:    end if
18: end for
```

Algorithm 10 shows how a thread t processes vertices in the local queue. First, an empty array `new_queue` is initialised, which is used to store neighbours of moved vertices. Thread t then iterates through the vertices in its local queue. The best move for each vertex is determined the same way as in the sequential algorithm. If vertex v is moved to a new community, the neighbours of v should be reprocessed. The neighbours of v are iterated through in line 6. A neighbour w should only be queued if it is not already in the queue and if the current community of w is the community v was moved to. The function CompareAndSwap is an atomic operation that will either set `in_queue[w] = true` and return `true` if w was not in the queue or change nothing and return `false` otherwise. This ensures that no thread places the same vertex in the queue twice. If neighbour w was not already in the queue, it is pushed to the array `new_queue`, and if `new_queue` reaches 1000 vertices, the array is pushed to the global queue to give waiting threads work. The size limit of 1000 was chosen to minimise unwanted overhead and idle time for threads. A low limit would lead to overhead from threads trying to acquire access to `global_queue` too often, and a high limit would lead to idle threads waiting for work for too long. After pushing the array to the global queue, thread t signals all waiting threads that new vertices were pushed and empties `new_queue`. When all neighbours of

v are processed, v will be set as not in the queue, and thread t will proceed to the next vertex in the local queue.

Algorithm 11 Waiting threads in the parallel local moving phase

```
 1: if new_queue $\neq \emptyset$ then                         ▷ Local work is available
 2:     local_Q $\leftarrow$ new_queue
 3:     continue
 4: end if
 5: if global_Q $\neq \emptyset$ then                          ▷ Global work is available
 6:     local_Q $\leftarrow$ global_Q.pop()
 7:     continue
 8: end if
 9: threads_waiting.fetch_add(1)                  ▷ Increment counter of waiting threads
10: if threads_waiting $<$ num_threads then              ▷ Some threads are still working
11:     Wait()                                                      ▷ Wait for work
12:     if threads_waiting $<$ num_threads then
13:         threads_waiting.fetch_sub(1)          ▷ Decrement counter of waiting threads
14:         if global_Q $\neq \emptyset$ then                         ▷ Work is available
15:             local_Q $\leftarrow$ global_Q.pop()       ▷ Try to take work from the global queue
16:         end if
17:         continue
18:     end if
19: end if
20: WakeAllThreads()                        ▷ Notify threads that they should stop working
21: break
```

Algorithm 11 shows what thread t does after the vertices in its local queue are processed. First, it will check if there are any vertices in the local array `new_queue`. If so, these are pushed to the local queue, and t will continue to the next iteration of the while loop shown in Algorithm 9. If `new_queue` is empty, the next step is to get work from the global queue. The global queue is synchronised, meaning that only one thread can pop an element from the queue simultaneously.

If both the local and global queues are empty, t will wait for more work to be pushed to the global queue by one of the other threads. First, the atomic variable `threads_waiting` will be incremented in line 9 to signal that it is idle. If t is not the only thread working, i.e. the statement on line 10 is `true`, it will wait to be signalled. At this point, two events can wake up t. If another thread has pushed work to the global queue, shown on line 11 in Algorithm 10, t will be signalled to wake up, decrement `threads_waiting`, pop work from the global queue, and continue to the next iteration in the while loop.

The second case where thread t will be woken is when all threads are done working. When a thread t' reaches line 10 and `threads_waiting` $=$ `num_threads`, it will jump to line 20 and signal all threads that there is no more work. Thread t, waiting in line 11, will then jump to line 20, and on the following line break the while loop.

5.2.2 Parallel Refinement

When implementing a parallel version of the refinement phase, it is important to remember why it was originally introduced. The refinement phase guarantees the absence of disconnected communities and increases connectivity within communities overall. These properties have to be upheld in the parallel version as well, guaranteed by these requirements:

- A vertex v has to be well-connected to its initial community S
- The target community C has to be well-connected.
- v has to be a singleton at the time of the move

In the sequential implementation, shown in Algorithm 6, the communities in $\mathcal{P}$ are processed iteratively. The most straightforward approach to parallelise the refinement phase would be to parallelise the for loop iterating through the communities. However, this will cause idle threads and uneven workload. If there are fewer communities in the partition than there are available threads, only $|\mathcal{P}|$ threads will work. Also, the sizes of the communities might be very different, meaning an uneven workload for the threads, probably resulting in idle threads.

To fully utilise the available threads, the vertices in G are iterated through instead of refining the partition community-wise. Each vertex is processed once, which means that iterating through the vertices in parallel will give a more even workload for each thread. However, this requires more synchronisation mechanisms to guarantee that the properties of the sequential refinement phase are upheld. The first requirement does not need to be synchronised since the initial community S is never changed during the refinement phase. Hence, two things must be true when vertex v is moved to community C: vertex v has to be in a singleton community, and C has to be well connected.

The problem is that both requirements might be fulfilled when the best community C is calculated but false when vertex v is moved. This problem is solved by locking the access to the current community of v and to community C, ensuring that no vertex can be moved to the community v is currently in, and no vertices can be moved away from C. Hence, when v moves, its current community is guaranteed to be a singleton, and community C is guaranteed to be well-connected.

Finding the best community for a vertex v is calculated the same way as in the sequential algorithm, by first finding all well-connected communities and then giving each community C' a probability of being chosen based on the increase in modularity of moving v to C', shown in lines 12 and 13 in Algorithm 6. Then, the thread will try to move v from its current community C to C' either until it succeeds or until another vertex w is moved to C, meaning v is not a singleton anymore and should not be moved, at which point the thread proceeds to the next vertex in the for loop.

Algorithm 12 Refinement phase of the parallel Leiden algorithm

```
1: function ParallelRefinePartition(Graph $G$, Partition $\mathcal{P}$)
2:     $\mathcal{P}_{\text{refined}} \leftarrow$ SingletonPartition($G$)        ▷ Assign each vertex to its own community
3:     parallel for $v \in V(G)$ do        ▷ Iterate through vertices of $G$
4:         if is_singleton[$v$] and $E(v, S - v) \geq \gamma\|v\| \cdot (\|S\| - \|v\|)$ then        ▷ Only consider well-connected vertices
5:             critical_vertices $\leftarrow \{w \mid w \in N_G(v)$ and $\mathcal{P}(v) = \mathcal{P}(w)\}$        ▷ Neighbours of $v$ in the same community
6:             Find new community $C'$
7:             LockLowerFirst($v, C'$)        ▷ Lock lower number first to avoid deadlock
8:             if is_singleton[$v$] then
9:                 while $C' = \emptyset$ do        ▷ Do not move to empty community
10:                    Unlock($v, C'$)        ▷ Unlock vertex $v$ and community $C'$
11:                    Update refined community weights
12:                    Find new community $C'$
13:                    if is_singleton[$v$] then
14:                        LockLowerFirst($v, C'$)
15:                    else
16:                        break        ▷ $v$ is no longer a singleton and should not be moved
17:                    end if
18:                end while
19:                if is_singleton[$v$] then
20:                    Update refined community weights
21:                    $v \mapsto C'$        ▷ Move vertex $v$ to community $C'$
22:                    is_singleton[$C'$] $\leftarrow$ false        ▷ $C'$ is no longer a singleton
23:                end if
24:            end if
25:        end if
26:        Unlock($v, C'$)        ▷ Unlock vertex $v$ and community $C'$
27:    end parallel for
28:    return $\mathcal{P}_{\text{refined}}$
29: end function
```

Three synchronised data structures are needed to avoid race conditions and deadlocks and to guarantee that the requirements are upheld. One array should define the total weight of the internal edges in the refined communities, and one should define the total weight of the external edges going out of the refined communities. These are used to decide whether the communities are well-connected.

The threads also need an array, `is_singleton`, of atomic booleans that determines whether the vertices are in singleton communities or are moved. All values are initialised as `true` and will be set to `false` as the vertices move. The last data structure needed is an array of locks, one for each community, used to block access to the current community of a vertex v and its chosen community C' to ensure the requirements hold.

In addition to the shared data structures, each thread has an array `critical_vertices`, consisting of the neighbours of the vertex v being processed, that are part of the same community as v. These are the only vertices affecting whether v should recalculate its best community.

Algorithm 12 shows the parallel implementation of the refinement phase of the Leiden algorithm. In line 4 thread t checks if vertex v should be processed, according to the requirements mentioned. Then, its neighbours in the same community are stored in `critical_vertices`. A community C' is chosen, shown in Algorithm 6. The communities are locked in line 7. A resource hierarchy [11] is used to avoid a deadlock, meaning the community with the lowest index is locked first.

There is some time from community C' is found to thread t acquires exclusive access to v and C'. In the meantime, two things can happen: a vertex w could have been moved to the community v is in, hence, v is not a singleton anymore, or, if C' was a singleton, the vertex in C' could have been moved, and C' is empty. If v is no longer a singleton, t will jump to line 26, release the locks and proceed to the next vertex. However, if v is still a singleton, t has to check that C' is not empty. In lines 9 through 18, thread t will repeatedly unlock v and C', update weight arrays, find new C' and lock v and C' as long as the chosen C' is empty. If v at any point is no longer a singleton, the while loop will break. When t exits the while loop, it has found a new community C' for v and locked both before anything changed. Vertex v can now be moved to C', which is no longer a singleton community, and the move meets both requirements.

5.3 Merging

The sequential implementation of the NCLiC algorithm uses a dynamic graph structure to store the already processed vertices and edges. The graph structure used in the parallel implementation is similar, except that it contains several synchronisation mechanisms to ensure memory safety and avoid race conditions.

The variables `n_vertices` and `n_edges` are implemented as `AtomicUsize` (2.8). This avoids race conditions when multiple threads try to increment the variables during the merging of the graphs. The array of adjacency lists `adj_lists` is a dynamic array, the same as in the non-thread-safe graph structure. However, the linked list at each index is protected by an `RwLock` (2.7). In the refinement phase, the adjacency lists are never written to but are often read by multiple threads, as seen on lines 8 and 23 in Algorithm 14. Hence, an `RwLock` will be efficient because it reduces the time threads wait to get read access. The array `vertex_weights` is never read or written to by any threads in the refinement phase, but to obey the rules of the ownership model, the data still has to be thread-safe. To ensure memory safety, the elements in the array are atomic variables. The array `id_to_label` ensures memory safety the same way as the `vertex_weights` array does, by protecting the variable at each index with atomics. The hash map `label_to_id` is protected using the third-party library `DashMap` [34]; a concurrent hash map that allows for thread-safe reads and writes of the values.

Algorithm 13 Graph merging phase of the parallel NCLiC algorithm

```
1:  function MergeGraph(Graph $G$, Graph $G_i$)
2:      $G$.n_edges ← +$G_i$.n_edges                          ▷ Increment number of edges
3:      parallel for $v \in V(G_i)$ do                       ▷ Prallel iterate through vertices of $G_i$
4:          vertex_label ← $G_i$.id_to_label[$v$]
5:          if vertex_label ∉ $G$.label_to_id.keys then       ▷ Only consider vertices not in $G$
6:              vertex_id ← $G$.n_vertices.FetchAdd()        ▷ Assign unique id to $v$ and increase the variable n_vertices
7:              $G$.label_to_id[vertex_label] ← vertex_id     ▷ Map label to id
8:              $G$.id_to_label[vertex_id] ← vertex_label     ▷ Map id to label
9:              $G$.vertex_weights[vertex_id] ← +$G_i$.vertex_weights[$v$] ▷ Increment vertex weight with weight of $v$ in $G_i$
10:         end if
11:     end parallel for

12:     parallel for $v \in V(G_i)$ do                       ▷ Iterate vertices of $G_i$
13:         vertex_label ← $G_i$.id_to_label[$v$]
14:         vertex_id ← $G$.label_to_id[vertex_label]         ▷ Get id of $v$ in $G$
15:         for $w \in N_{G_i}(v)$ do                         ▷ Iterate neighbours of vertex $v$ in $G_i$
16:             neighbour_label ← $G_i$.id_to_label[$w$]
17:             neighbour_id ← $G$.label_to_id[neighbour_label]   ▷ Get id of $w$ in $G$
18:             $G$.adj_lists[vertex_id].push(neighbour_id)   ▷ Add neighbour to adjacency list
19:         end for
20:     end parallel for
21: end function
```

Algorithm 13 shows the implementation of the merging of graph G_i into graph G, occurring in two steps. First, the vertices are merged into the graph. Only vertices that are not already in G should be considered; hence, the threads check on line 5 that the original labels of the vertices are not in the mapping from label to ids. If a vertex v should be merged into G, it is assigned an id, and because the vertices in G are zero-indexed, G.`n_vertices` will be the next unique id. To avoid a data race, this is an atomic variable. When the function FetchAdd() is called on this variable, it will be atomically incremented, and the previous value will be assigned to v. Hence, all vertices not already part of G will be assigned a unique id. Then, G.`id_to_label` and G.`label_to_id` will be updated, and G.`vertex_weights` will be incremented with the weight of v in G_i. From the guarantee that `vertex_id` and `vertex_label` are unique values, it follows that a data race cannot be introduced in this step.

When all vertices are added, the edges should be added. The vertices in G_i are iterated through in parallel. When a thread t processes a vertex v, it will first get the label of v from G_i in line 16, then get the id of v in G from the label in line 17. Then, for each neighbour w of v, thread t will obtain the id of w in G and push w to the adjacency list of v. As mentioned in section 5.4, iterating through the neighbours of all vertices might result in uneven workload among the threads, and thus iterating through the edges might be better. However, because the variable `adj_lists` is protected by an `RwLock`, repeatedly acquiring and releasing the lock will lead to unwanted overhead. When iterating through the neighbours of v, only thread t will have to acquire the lock. This can be done before the for loop in line 15, and released when all neighbours are pushed to the adjacency list, in line 19.

5.4 Refinement

In the sequential implementation of the NCLiC algorithm, the refinement step iterates through the vertices in the new graph G_i and for each vertex v, it performs four steps. The first step is to initialise NCC, if the vertex has not been part of an earlier chunk. The next step is to update the neighbour community count NCC with the new communities of the neighbours of v in G_i. The third step is to decide whether v should move to another community. The last step only occurs if the algorithm decides that v should move. If so, v is moved, and the neighbour community count of its neighbours might be updated based on the degree of v. In the sequential implementation, all four steps are applied to a vertex v before proceeding to the next vertex.

The steps in the parallel implementation are the same as in the sequential one, but they occur in a different order. To avoid race conditions, each step has to be applied to all vertices before proceeding to the next step. This means that the parallel refinement phase has to be split into four separate steps, and synchronisation mechanisms must be introduced for the data structures. In addition to the data structures already mentioned in Section 3.3.1, the parallel refine phase needs an array, `new_communities`, to store the new communities chosen for the vertices between the second and third steps.

Two data structures are read and written to in the refinement phase and must have synchronisation mechanisms to ensure memory safety and obey the rules of the ownership model. Multiple threads access the array NCC in the first and last steps. Accessing two separate indices simultaneously will not introduce a race condition, but two threads can not be allowed to write to a hash map at the same index simultaneously. Because the hash maps are more often written to than read, they are protected using a `Mutex`. This ensures mutually exclusive access to the hash maps when reading and writing.

The implementation of partition $\mathcal{P}$ has to ensure memory safety as well. In the first step, $\mathcal{P}$ is read by multiple threads concurrently and in the last step, $\mathcal{P}$ is read and written to by multiple threads. $\mathcal{P}$ is represented by an array where each element is implemented as an atomic variable to ensure exclusive access to read and update the values.

Even though the array `new_communities` is written to by multiple threads, each index will only be written to by one thread once and, in a separate step, be read by one thread once; hence, no race condition will occur. Because the ownership model can verify during compile time that no race conditions will occur, the array does not need extra synchronisation mechanisms.

Algorithm 14 Refinement phase of the parallel NCLiC algorithm

```
 1: function RefineCommunities(Partition $\mathcal{P}$, Graph $G$, Graph $G_i$, HashMap $NCC$)
 2:     new_communities ← empty array                    ▷ Initialise empty array to store new communities
 3:     parallel for $v \in V(G_i)$ do
 4:         if $v \notin V(G)$ then
 5:             $NCC[v] \leftarrow$ empty hash map          ▷ Initialise neighbour community count for new vertex $v$
 6:         end if
 7:     end parallel for

 8:     parallel for $\{v, w\} \in E(G_i)$ do              ▷ Iterate through edges in $G_i$ in parallel
 9:         $NCC[v][\mathcal{P}(w)] \leftarrow +1$          ▷ Increment count of the community of neighbour $w$ of $v$
10:         $NCC[w][\mathcal{P}(v)] \leftarrow +1$          ▷ Increment count of the community of neighbour $v$ of $w$
11:     end parallel for

12:     parallel for $v \in V(G_i)$ do
13:         new_communities[$v$] ← $\arg\max(NCC[v])$      ▷ Find the community that most neighbours of $v$ are assigned to
14:     end parallel for

15:     parallel for $v$ in $V(G_i)$ do                      ▷ Iterate through vertices in $G_i$ in parallel
16:         $C_{\text{old}} \leftarrow \mathcal{P}(v)$
17:         $C_{\text{new}} \leftarrow$ new_communities[$v$]
18:         if $C_{\text{old}} \neq C_{\text{new}}$ then
19:             $v \mapsto C_{\text{new}}$                  ▷ Move $v$ to $C_{\text{new}}$
20:             $x \leftarrow$ avgdegree − degree($v$)
21:             $p \leftarrow \left(\frac{x}{\sqrt{1+x^2}} + 1\right)/2$   ▷ Calculate the probability that the $NCC$ of neighbours should be updated
22:             if random $< p$ then
23:                 for $w \in N_G(v)$ do                   ▷ Update $NCC$ for all neighbours of $v$
24:                     $NCC[w][C_{\text{old}}] \leftarrow -1$    ▷ Decrease count of old community
25:                     $NCC[w][C_{\text{new}}] \leftarrow +1$    ▷ Increase count of new community
26:                 end for
27:             end if
28:         end if
29:     end parallel for
30: end function
```

Algorithm 14 shows the refinement phase of the parallel NCLiC algorithm. In line 2, the empty array `new_communities` is initialised. The first step is to initialise the neighbour community count NCC for the new vertices $V(G_i) \setminus V(G)$, in line 5.

The next step is to update the neighbour community count for all the vertices in G_i. In the sequential implementation, the neighbours of the vertex v currently being processed are iterated over to count the communities adjacent to v. However, because the degree of the vertices may differ, this approach may distribute the work unevenly among the threads. To update the neighbour community count of all vertices, this step requires that all neighbours of all vertices are iterated over and their communities counted. This is the same as iterating through all the edges in G_i, as seen in line 8. For each edge $\{v, w\}$, the neighbour community count of vertex v and w is incremented. This approach distributes the work more evenly among all the threads.

The third step is to compute the most common community among the neighbours for all vertices and store the results in the array `new_communities`, seen in line 13. This step has to be separated from the last step because, in the last step, the neighbour community count is updated several times and would definitely introduce a race condition when reading and updating the values simultaneously.

The last step is to move the vertices to their chosen communities. The threads iterate concurrently through the vertices in G_i. For each vertex v, thread t checks whether the current community of v is the same as the community chosen in the previous step in line 18. If they differ, v will be moved to its new community. Then, t will compute the probability that the NCC of the neighbours of v should be updated, explained in Section 3.3.2. The neighbours of v are then iterated through in line 23, and the NCC of each neighbour is updated. Iterating through the neighbours might result in an uneven workload for the threads as a result of the difference in the degrees of the vertices. However, because the probability of whether the update should happen is decreasing with an increasing degree of v, the uneven workload among the threads is insignificant.

5.5 Summary

In this chapter, we have presented an approach to parallelise the NCLiC algorithm. For the pre-clustering, a parallel implementation of the Leiden algorithm is used. All steps of the algorithm are parallelised, with a specific focus on the fast local moving and refinement phases. The refinement phase of the NCLiC algorithm is parallelised by reordering the steps applied to the vertices. This enables the threads to process the vertices simultaneously and ensures an even workload. In the merging phase, the vertices of the arriving chunk are merged into the larger graph in parallel, followed by the parallel merging of all the edges.

Chapter 6

Experiments

In this chapter, we explain how the tests were run, on which datasets and why, the hardware used to run the tests, and the experimental setup for the tests. The results of the tests are presented and we discuss which factors affect the efficiency of the algorithm.

6.1 Data Sets

To test the sequential and parallel implementations of the Leiden algorithm, tests were run on the DIMACS10 graph set [2], consisting of 151 graphs. The DIMACS10 challenge considered the problems of graph clustering and graph partitioning, and the data set is designed to be a standardised benchmark for these problems. The graphs vary in size, with the smallest having 39 vertices and 340 edges and the largest having over 16 million vertices and 265 million edges. The data set also consists of some multigraphs, but due to the fact that the NCLiC algorithm is not meant for these types of graphs, they were not considered, resulting in a total of 145 graphs used for our tests. When clustered by the sequential Leiden algorithm, 115 graphs have a modularity of more than 0.9, 16 graphs have a modularity between 0.8 and 0.9, and 14 graphs have a modularity of less than 0.8.

The sequential NCLiC algorithm was run on the SNAP Twitter graph [21] to compare its performance to that of the original Python implementation by Tumanis [31, p. 89]. The algorithm was tested on the DIMACS10 data set as well to analyse the average modularity retention of the sequential NCLiC algorithm compared to the Leiden algorithm and the average modularity retention of the parallel NCLiC algorithm compared to the sequential implementation. The scaling of the parallel NCLiC algorithm was also tested on a very large graph: GAP-web [5] with $|V| = 50$ mill. and $|E| = 9,3$ bill. The GAP Benchmark Suite is used as a standardised benchmark for graph algorithms and was chosen for the experiments in this thesis due to the size of the graphs.

6.2 Hardware

The experiments were run on the Simula eX3 super computer [14] on a dual processor node with the AMD EPYC Milan 7763 64-core processor with a base clock speed of 2.45 GHz and a total of 256 threads using multithreading. The node has 2 TB DDR4 main memory and 7.6 TB GB local NVMe scratch storage. The source code was compiled using the rustc compiler version 1.67.0.

6.3 Experimental Setup

The Leiden algorithm was run on all graphs in the DIMACS10 data set to test parallel scaling and modularity retention of the parallel implementation compared to the sequential implementation. The algorithm was run for two iterations, with a resolution of 1.0 and a randomness of 0.01. These settings were used both for the separate Leiden tests and when used for pre-clustering in the NCLiC algorithm. To make the tests reproducible, all tests were run with the same parameters and without shuffling the data sets.

The NCLiC algorithm was tested on several graphs, sliced into k chunks. The first chunk was set to be 20% of the total number of edges in the graphs, and the rest of the edges were evenly divided among the $k - 1$ remaining chunks. The choice of the size of the initial chunk is discussed in [31, pp. 90-91]. The tests were run with k increasing by a power of two, $k = 1, 2, 4, ..., 1024, 2048$, to be able to detect changes when k was small and discover trends in modularity retention and scalability as k grew.

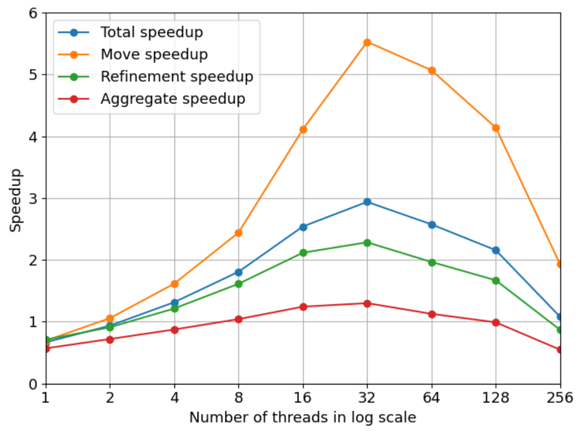
6.4 Results

In this section, we present and discuss the results of the parallel Leiden algorithm, the sequential NCLiC algorithm and the parallel NCLiC algorithm. We present the execution times of the algorithms and the speedups achieved compared to the sequential implementations. To evaluate the quality of the partitions discovered by the algorithms, we compare the modularity achieved by the parallel algorithms to that of the sequential implementations and measure the percentage of the modularity the parallel implementations retains, hereafter referred to as *modularity retention*.

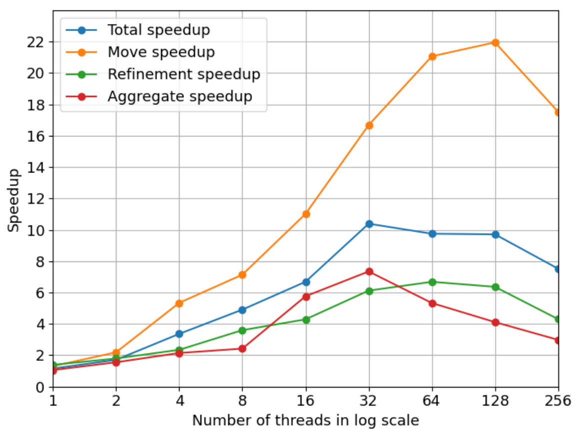
6.4.1 Parallel Leiden

DIMACS10 Graphs

First, we present the results of the parallel implementation of the Leiden algorithm used in the pre-clustering phase of the NCLiC algorithm. The algorithm was tested on the DIMACS10 data set with up to 256 threads. As the DIMACS10 graphs differ significantly in both size and optimal modularity, a good way to analyse the performance of the algorithm on this data set is to cluster each graph in the data set with the sequential and the parallel Leiden algorithm and look at the average speedup of the graphs and the average modularity retention.



(a) Average speedup



(b) Maximum speedup

Figure 6.1: Speedup of the parallel Leiden algorithm on DIMACS10

Figure 6.1 shows the speedup of the parallel Leiden algorithm compared to the sequential implementation on the graphs in the DIMACS10 data set. Figure 6.1a shows the average speedup of the three phases of the algorithm, and the total speedup. As shown, the local moving obtained the most speedup with up to an average factor of 5.5, resulting in an average total speedup of 2.9 compared to the sequential implementation.

Because the speedup of the algorithm is highly dependent on the size of the graph being clustered and the DIMACS10 data set consists of several small graphs, we included the maximum speedups as well, shown in Figure 6.1b. In this plot, the local moving phase achieves a speedup of up to a factor 21.9 with a maximum total speedup of up to a factor of 10.3 compared to the sequential Leiden algorithm.

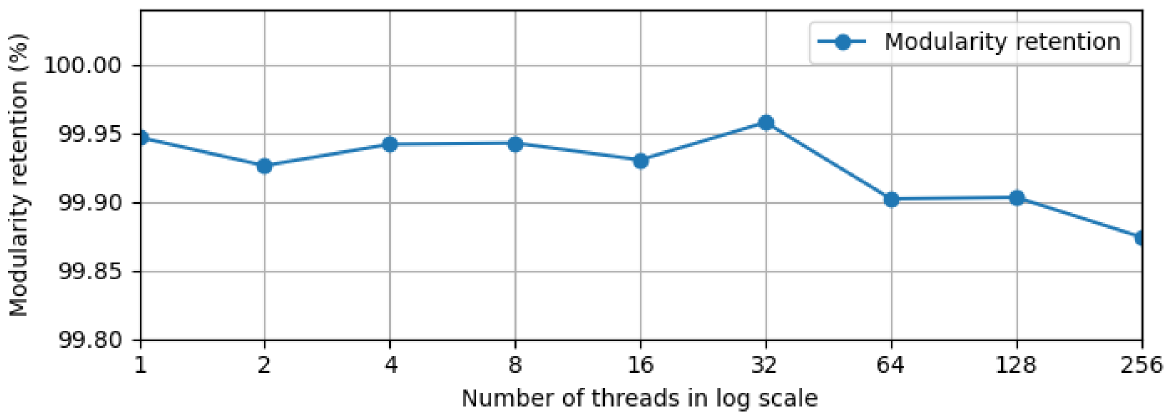


Figure 6.2: Modularity retention of the parallel Leiden algorithm on DIMACS10

Figure 6.2 shows the average modularity retention achieved by the parallel Leiden algorithm on the DIMACS10 graphs. The average modularity retention stays above 99.8% for all threads. For 97.5% of the tests, the resulting modularity retention was over 99.5%, and 90% of the tests resulted in more than 99,9% modularity retention.

Random Geometric Graphs

Figure 6.3 shows the execution time of each number of threads as the size of the graph increases, run on the random geometric graphs from the DIMACS10 data set. The execution times can give an indication of the scalability of the algorithm, as the time used for a lower number of threads clearly increases faster than for the higher numbers. However, using 64 and 128 threads, the execution times are approximately the same for all numbers of vertices.

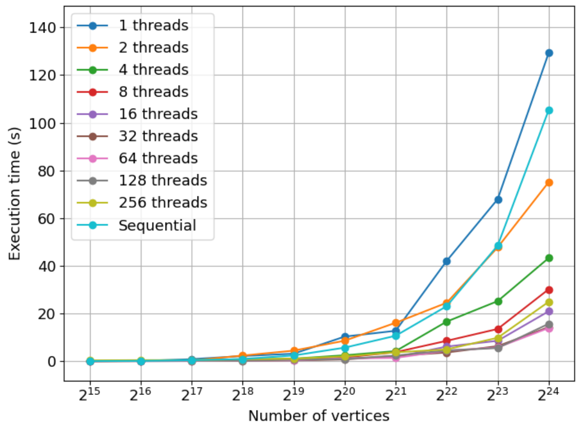
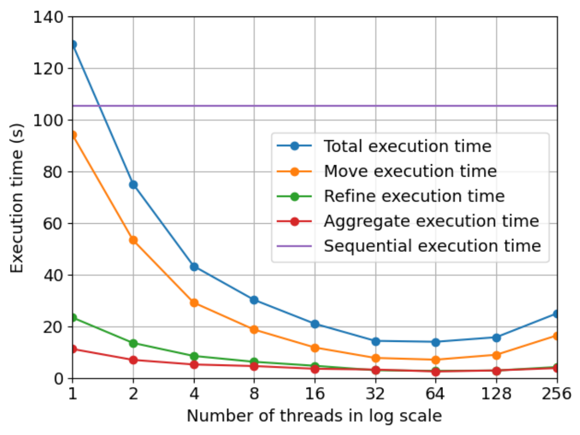
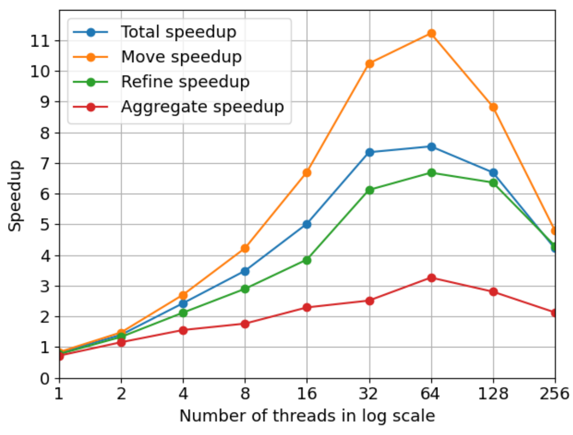


Figure 6.3: Execution time of parallel Leiden algorithm on random geometric graphs



(a) Execution time versus number of threads



(b) Speedup versus number of threads

Figure 6.4: Speedup of the parallel Leiden algorithm on rgg_n_2_24_s0

Figure 6.4 shows the execution time and speedup of the parallel Leiden algorithm on the largest graph in the DIMACS10 data set: rgg_n_2_24_s0, with 2^{24} vertices. Figure 6.4a shows how the execution time is distributed among the three phases of the algorithm as the number of threads increases. The sequential execution time is included. With one thread, the local moving phase takes around three times longer than the refinement phase and nine times longer than the aggregation phase. However, for graphs of this size and structure, the local moving scales better than the other phases, with a speedup of up to a factor of 11, shown in Figure 6.4b. The refinement phase and aggregation phase achieved speedups up to 6.6 and 3.3, respectively. The maximum speedup of the whole algorithm is 7.6, using 64 threads. The modularity retention of the parallel Leiden algorithm on rgg_n_2_24_s0 stays above 99.997% for all thread configurations.

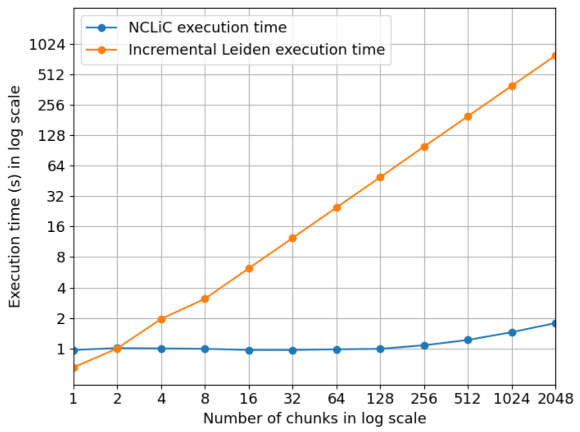
The parallel Leiden algorithm proved to be effective on large graphs and loses virtually no modularity compared to the sequential implementation. An important factor that affects the efficiency of the algorithm is the structure of the graph being clustered, as clustering graphs with a high density turned out to give more speedup compared to sparser graphs of the same size. This is, however, merely an observation and has not been tested thoroughly in this thesis.

6.4.2 Sequential NCLiC

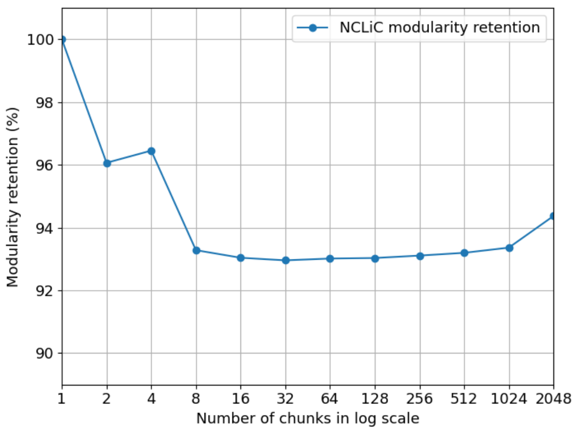
Here we present a comparison of the sequential NCLiC algorithm and the Incremental Leiden algorithm, that is, running Leiden each time a new chunk is merged into the graph. We also look at the quality of the partitions discovered by the sequential NCLiC algorithm, compared to the partitions the Leiden algorithm found.

Incremental Leiden

Figure 6.5 shows a comparison of the NCLiC and the Incremental Leiden algorithms.



(a) Execution times



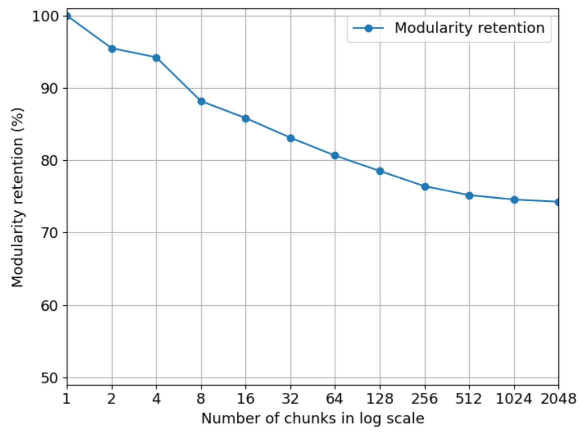
(b) Modularity retention

Figure 6.5: Incremental Leiden versus NCLiC on SNAP Twitter

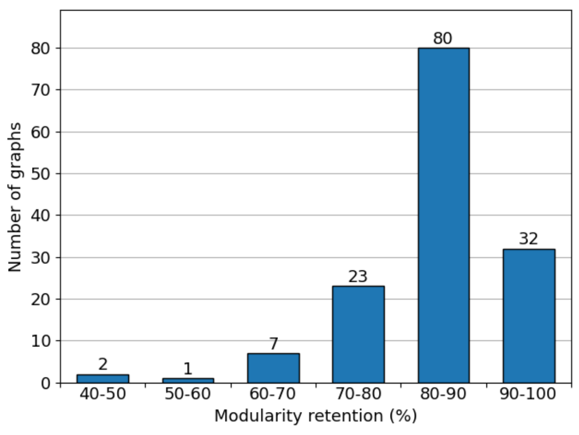
Figure 6.5a shows the execution times of the algorithms when the number of chunks increases. Note that the y-axis is in logarithmic scale. The sequential NCLiC algorithm clearly uses a significantly shorter time than the Incremental Leiden algorithm. However, the cost of the reduced execution time is a reduction in modularity. Figure 6.5b shows the modularity retention of the NCLiC algorithm, compared to the modularity that Leiden achieved. The modularity quickly drops to around 93%, but after that, it flattens out and increases slightly as the number of chunks increases.

DIMACS10 Graphs

The modularity achieved by NCLiC differs from graph to graph, depending on the size and structure of the graph, and the size of the chunks. Figure 6.6a shows the average modularity retention of the graphs in DIMACS10 when the number of chunks increases. The average modularity retention slowly decreases when the number of chunks increases and seems to flatten as the number of chunks approaches 2048. Figure 6.6b shows the distribution of modularity retentions of the graphs in DIMACS10. The modularity retention for each graph is the average modularity retention of all chunks from one to 2048, increasing by a power of two. More than 93% of the DIMACS10 graphs retained a modularity higher than 70%, and over 22% of the graphs had a modularity retention of 90 to 100%.



(a) Average modularity retention



(b) Frequency of modularity retention

Figure 6.6: Modularity retention of NCLiC on DIMACS10

6.4.3 Parallel NCLiC

In this last section, we present the results of the parallel NCLiC algorithm. First, we present the results of running the algorithm on the DIMACS10 data set, followed by an in-depth analysis of the results from the tests run on the GAP-web graph.

DIMACS10 Graphs

Figures 6.7 through 6.9 shows the average modularity achieved by the parallel NCLiC algorithm on the DIMACS10 graphs.

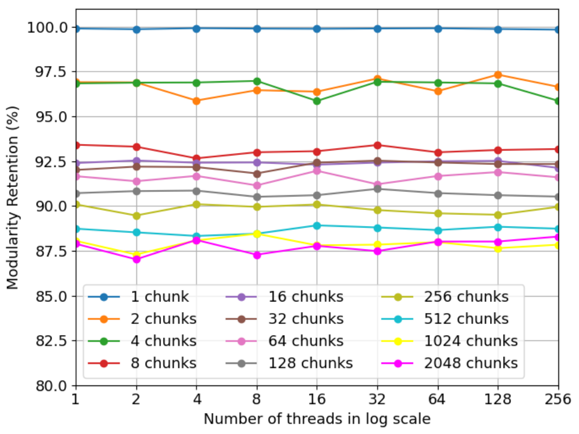


Figure 6.7: Modularity retention versus threads of parallel NCLiC on DIMACS10

Figure 6.7 shows the modularity retention for each number of chunks, with an increasing number of threads. Not surprisingly will a lower number of chunks, in general, result in higher modularity retention than a higher number of chunks. The interesting thing to note is that, as the number of threads increases, the modularity retention remains at approximately the same level for all numbers of chunks, which gives an indication that the number of threads does not have much impact on the modularity retention.

Figure 6.8 shows the modularity retention of each thread configuration as the number of chunks increases. As seen in Figure 6.7, the number of threads does not affect the modularity significantly; hence, the lines are mostly overlapping. The modularity retention of all thread configurations starts at 100% and steadily decreases as the number of chunks increases, down to around 88% for 2048 chunks. It is, however, important to remember that this is the average of all the graphs and that the modularity retention differs depending on the size and structure of the graph.

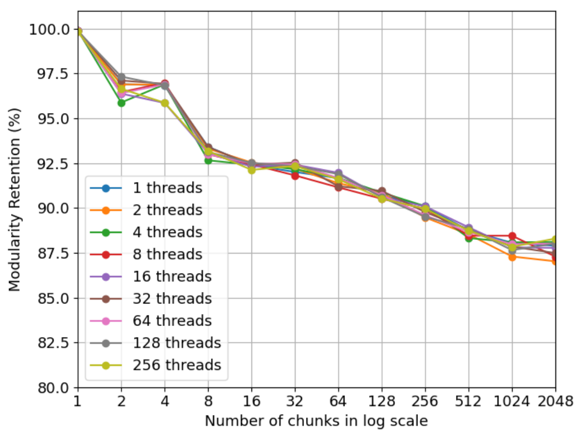


Figure 6.8: Modularity retention versus chunks of parallel NCLiC on DIMACS10

Figure 6.9 shows the distribution of modularity retentions of the graphs in DIMACS10. The modularity retention of each graph is the average of all chunks from one to 2048 and all threads from one to 256, both increasing by a power of two. More than 90% of the graphs have modularity retention higher than 80%, and more than 79% of the graphs retained a modularity higher than 90% compared to the sequential implementation.

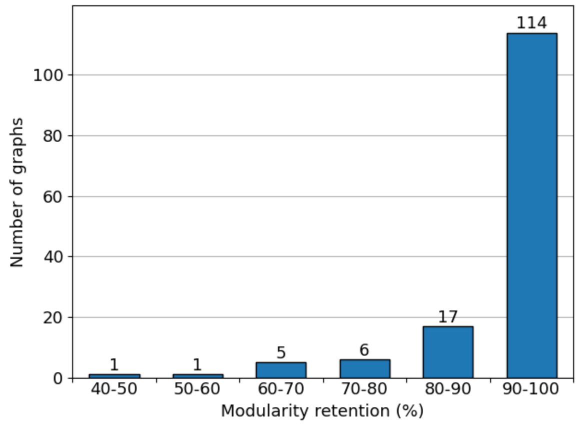


Figure 6.9: Distribution of modularity retention of parallel NCLiC on DIMACS10

Figure 6.10 shows the execution times of the sequential and parallel NCLiC algorithm for various numbers of threads on the random geometric graphs, with 16 chunks. As seen in Figure 6.3, the parallel Leiden algorithm is more efficient for larger graphs. The same pattern can be seen for the parallel NCLiC algorithm, with the execution times with eight or more threads increasing slower than the sequential execution time, as the size of the graph increases.

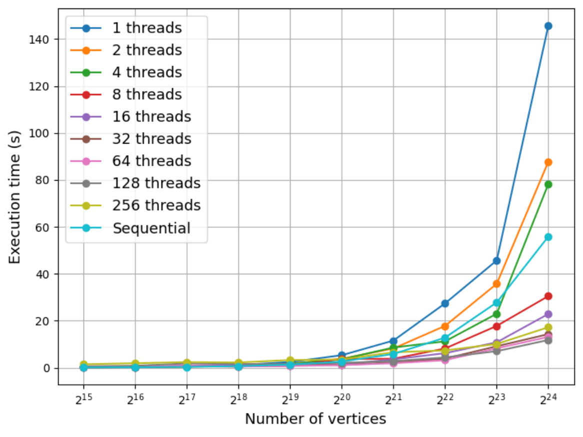
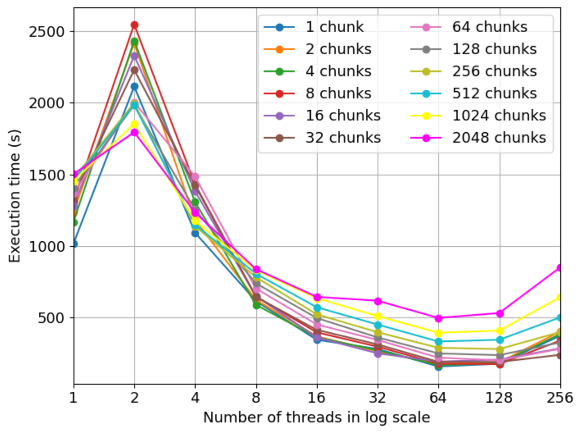


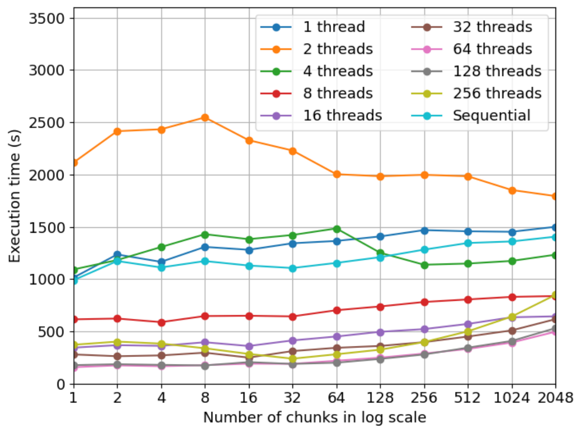
Figure 6.10: Execution time of parallel NCLiC on random geometric graphs

GAP-web Graph

Figures 6.11 through 6.13 show the execution time, speedup and modularity retention of the parallel NCLiC algorithm on the GAP-web graph.



(a) Execution time versus threads

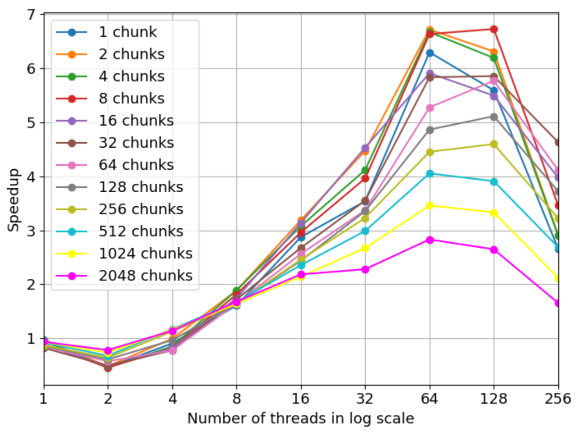


(b) Execution time versus chunks

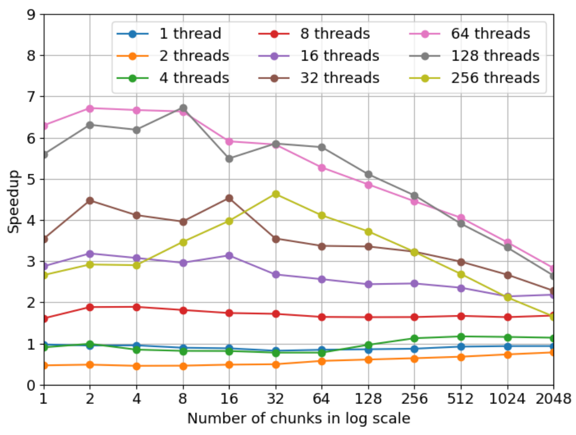
Figure 6.11: Execution time of parallel NCLiC on GAP-web

Figure 6.11a shows the execution time of running the algorithm with an increasing number of threads for different numbers of chunks. The execution times follow roughly the same pattern as the number of threads increases. For a low number of threads, a higher number of chunks tends to be faster than for a low number of chunks. However, when the number of threads increases, the execution time of the higher number of chunks is more than twice the execution time for the lower number of threads. This could be a result of the threads trying to acquire access to data structures, and with smaller chunks, there is a higher probability they try to acquire locks simultaneously.

Figure 6.11b shows the execution time of each thread as the number of chunks increases. The execution time of the sequential NCLiC algorithm is also included. Most of the execution times follow the same pattern, where they increase slightly with the number of chunks. The plot clearly shows that for all numbers of chunks, the execution times from using eight up to 256 threads are lower than the sequential one. The execution of the parallel implementation using one thread is slightly slower than the sequential implementation. With four threads, the execution is slower up to 128 chunks, where it becomes slightly faster. The execution using two threads is slower for all numbers of chunks. This is also reflected in Figure 6.11a, where the execution times peak significantly with two threads.



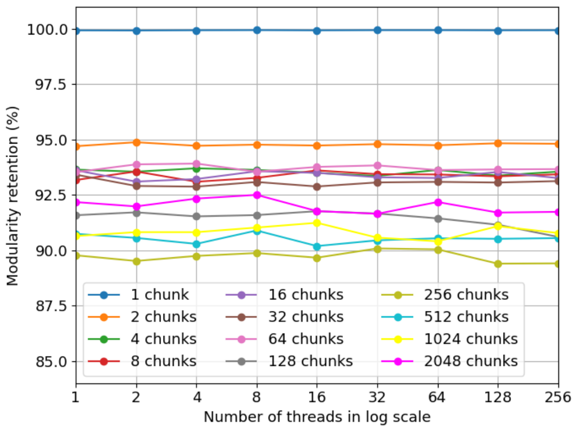
(a) Speedup versus threads



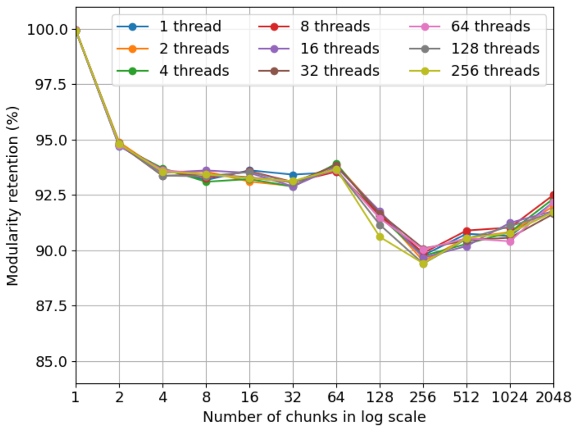
(b) Speedup versus chunks

Figure 6.12: Speedup of parallel NCLiC on GAP-web

Figure 6.12a shows the speedup of the algorithm for different numbers of chunks with an increasing number of threads. This plot shows that the number of chunks, or more precisely, the size of each chunk, affects the speedup significantly, which is probably affected by the fact that the speedup of Leiden is very dependent on the size of the graph being clustered. The trend can also be seen in Figure 6.12b, where the number of threads with the most speedup decreases as the number of chunks increases.



(a) Modularity retention versus threads



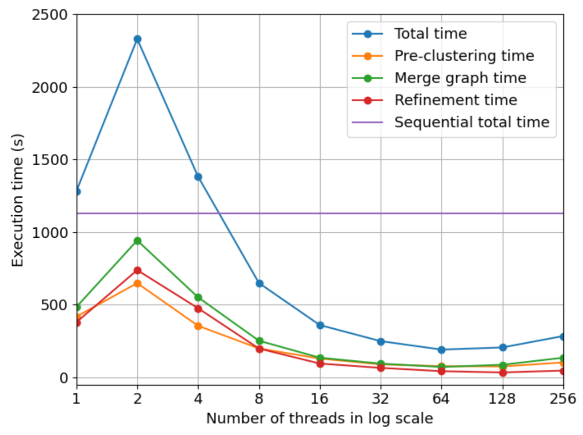
(b) Modularity retention versus chunks

Figure 6.13: Modularity retention of parallel NCLiC on GAP-web

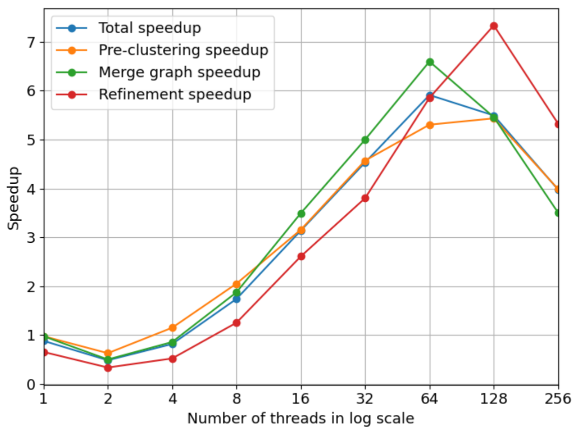
Figure 6.13a shows the modularity retention of the parallel NCLiC algorithm compared to that of the sequential NCLiC algorithm for each number of chunks, with an increasing number of threads. These results show the same pattern as the average of the DIMACS10 graphs did in Figure 6.7, with a lower number of chunks giving higher modularity retention than a higher number of chunks, and with a relatively stable level of modularity retention as the number of threads increases. For this particular graph, the modularity retention is above 88% for all numbers of chunks.

Figure 6.13b shows the modularity retention of each thread configuration as the number of chunks increases. As seen in the previous plot, the number of threads does not affect the modularity significantly; hence, the lines are mostly overlapping. For all threads, the modularity drops to around 94% of the modularity obtained from the sequential implementation for up to 64 chunks. It then drops down to 90% with 256 chunks and, in contrast to the average for the DIMACS10 graphs in Figure 6.8, the modularity retention increases slightly up to 2048 chunks.

The results of running the parallel NCLiC algorithm on the GAP-web graph show some trends, both in terms of execution time and modularity retention. First of all, we can conclude that the number of threads used to run the algorithm has little to no effect on the modularity score. The most important factor that affects the modularity is the number of chunks used when running the algorithm, which can be explained by the fact that the Leiden algorithm is able to perform a better local pre-clustering with larger chunks, and the refinement phase of NCLiC will not have to merge the partitions as often. This trend could also be seen when comparing the sequential NCLiC algorithm to the Leiden algorithm in Section 6.4.2.



(a) Execution time



(b) Speedup

Figure 6.14: Execution time and speedup of phases in parallel NCLiC on GAP-web

Figure 6.14 shows the execution time and speedup of the different phases of the parallel NCLiC algorithm for 16 chunks. Figure 6.14a shows the execution time for each phase, including the sequential execution time. An interesting note is that all three phases take up a significant amount of the total execution time, showing the importance of parallelising and speeding up all phases. The execution times follow the pattern seen in Figure 6.11a, where the execution time peaks with two threads and then decreases as the number of threads increases. Figure 6.14b shows the speedup of the different phases. As mentioned, the execution time of both the sequential and parallel implementation depends heavily on the size of the chunks being processed. Hence, the speedup of the different phases varies from chunk size to chunk size. For 16 chunks, the pre-clustering and refinement phase scales up to 128 threads. The merging of the graphs scales up to 64 threads, along with the execution of the whole algorithm.

Chunks	**Pre-clustering**		**Refinement**		**Merge graphs**		**Total**	
	Speedup	Threads	Speedup	Threads	Speedup	Threads	Speedup	Threads
1	8.48	64	-	-	5.27	64	6.3	64
2	8.91	128	18.42	128	4.64	64	6.72	64
4	7.91	64	12.53	128	4.82	64	6.67	64
8	6.9	64	9.74	128	6.24	64	6.73	128
16	5.43	128	7.49	128	6.6	64	5.91	64
32	4.47	64	6.59	128	8.37	64	5.86	128
64	3.94	64	5.71	128	10.36	128	5.77	128
128	3.72	64	5.19	128	9.45	128	5.11	128
256	3.65	64	4.38	128	8.35	128	4.6	128
512	3.45	64	3.46	128	6.48	128	4.05	64
1024	3.07	16	2.92	128	5.77	128	3.46	64
2048	2.9	16	2.25	128	4.63	128	2.83	64

Table 6.1: Maximum speedup of phases in parallel NCLiC on GAP-web

Table 6.1 shows the maximum speedup for each phase for all numbers of chunks, together with the number of threads used to achieve the speedup. The pre-clustering achieves a speedup of 8.91 using two chunks and decreases down to 2.9 as the number of chunks increases. A drop in speedup as the number of chunks increases is expected, as the efficiency of the parallel Leiden algorithm is very dependent on the size of the graph being clustered. The pre-clustering phase scales up to 64 threads for most chunks, but for the two highest numbers of chunks, the pre-clustering only scales up to 16 threads.

The efficiency of the refinement phase is clearly affected by the size of the chunks, as it starts with a speedup of 18.42 for two chunks, which decreases steadily down to 2.25 for 2048 chunks. It does, however, scale up to 128 threads, the maximum number of threads without using multithreading, for all numbers of chunks. Same as for the pre-clustering phase, less work per chunk results in more overhead in total. Note that for one chunk, the refinement phase is not used; hence, it is blank.

The merging of the graphs also shows a clear dependency on the size of the chunks, with an increasing speedup from 4.64 for two chunks up to 10.36 for 64 chunks, with 128 threads, and then decreases down to 4.63 for 2048 chunks. In contrast to the other two phases, the speedup of the merging phase increases up to 64 chunks, which is probably the best number of chunks to minimise overhead from locking data structures while still processing vertices in parallel effectively.

The total speedup peaks at a factor of 6.73 compared to the sequential implementation, with 8 chunks and 128 threads. From 8 chunks, the speedup slowly decreases to 2.83 for 2048 chunks.

Chapter 7

Conclusion and Future work

7.1 Conclusion

With the increasing number of complex real-world networks, there is a corresponding need for improved data analysis and pattern discovery. Community detection can yield valuable information about the underlying structure of graphs, revealing their organisational principles and functional mechanisms. However, traditional algorithms struggle to keep up with the scale and complexity of incremental graphs, often leading to sub-optimal solutions or unviable computational demands. Parallel computing emerges as a promising solution to this scalability issue.

In this thesis, we explored different strategies to parallelise the NCLiC algorithm. The algorithm consists of four steps, repeated for each chunk of data that arrives: building a graph of the chunk, pre-clustering the chunk, refining the partition produced by the pre-clustering and merging the chunk into the graph consisting of already processed vertices and edges. We introduced a novel parallel version of the NCLiC algorithm, where all steps except for the graph building are parallelised, using a shared memory approach.

The algorithms were evaluated through extensive testing. The algorithms were tested on the DIMACS10 data set to discover trends in the algorithm on different types of graphs. In addition, the parallel NCLiC algorithm was tested on the GAP-web graph, with 50.6 million vertices and 1.9 billion edges, to discover trends in scaling and modularity retention on a very large graph.

A parallel implementation of the well-known Leiden algorithm is used for the pre-clustering phase. The parallel Leiden algorithm showed promising results, with virtually no loss in modularity: an average of 0.1% for the DIMACS10 graphs and 0.003% for the largest of the DIMACS10 graphs. The algorithm obtained speedups up to a factor of 10.3 compared to the sequential implementation. When used in the pre-clustering phase of the NCLiC algorithm on the GAP-web graph, the Leiden algorithm scaled up to 128 threads.

The refinement phase of the parallel NCLiC algorithm is similar to the sequential implementation but with some modifications to ensure memory safety and to effectively utilise all available threads. Instead of processing vertices one by one, the algorithm is split into four separate steps to be able to process all vertices simultaneously. The refinement phase of the NCLiC algorithm obtained speedups up to a factor of 18.42, dependent on the size of the chunk being processed, scaling up to 128 threads for all numbers of chunks.

The merging of the chunk into the main graph was also parallelised by first adding the vertices to the graph in parallel, followed by the edges. The merging of the graphs obtained speedups up to a factor of 10.36, scaling up to 64 threads with up to 32 chunks and up to 128 threads for chunk sizes higher than 32.

The whole parallel NCLiC algorithm obtained speedups up to a factor of 6.73 with 128 threads. The number of threads showed to have little to no effect on the modularity retention, both for the average of the DIMACS10 graphs and the GAP-web graph. The modularity retention was, however, affected by the number of chunks, the same trend seen for the sequential NCLiC algorithm when compared to the modularity of the Leiden algorithm, and is probably also dependent on the structure of the graph.

Throughout this thesis, we have explored possible approaches and established some requirements for parallelising the NCLiC algorithm. We have provided an algorithm that achieves significant speedup but also discovered some challenges that can lay the foundation for further research. My hope is that the work presented here may be useful for potential future research in the field of community detection and graph theory.

7.2 Future Work

In the pre-clustering phase, a parallel implementation of the Leiden algorithm is used. The NCLiC algorithm is, however, not bound to using this algorithm to pre-cluster the incoming chunks. Exploring the possibility of using optional algorithms for community detection could result in increased performance in terms of efficiency, scalability and quality. The same applies to the merging of the chunk into the larger graph, as other graph structures might prove to be more efficient and scalable.

The quality of the communities detected is measured using modularity, but same as for the pre-clustering phase, the NCLiC algorithm is not bound to this measure. Several quality measures exist, for example, the Constant Potts Model [30]. Testing different quality functions might result in increased quality and efficiency, both sequentially and parallel.

The algorithm was tested on the DIMACS10 data set and the GAP-web graph to get an indication of the performance of the algorithm in terms of efficiency, scalability and modularity retention. Testing the algorithm on even larger and more graphs could, however, provide improved results and might give an even deeper understanding of the strengths and weaknesses of the algorithm.

List of Acronyms and Abbreviations

API	application programming interface.
NCLiC	Neighbourhood-to-Community Link Counting.

Bibliography

[1] Amdahl, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, 1967.
URL: `https://doi.org/10.1145/1465482.1465560`.

[2] Bader, D. A., Meyerhenke, H., Sanders, P., and Wagner, D. *Graph partitioning and graph clustering*, volume 588. American Mathematical Society Providence, RI, 2013.
URL: `https://sparse.tamu.edu/DIMACS10`.

[3] Balasubramanian, A., Baranowski, M. S., Burtsev, A., Panda, A., Rakamarić, Z., and Ryzhyk, L. System programming in rust: Beyond safety. In *Proceedings of the 16th workshop on hot topics in operating systems*, pages 156–161, 2017.
URL: `https://doi.org/10.1145/3102980.3103006`.

[4] Blondel, V. D., Guillaume, J.-L., Lambiotte, R., and Lefebvre, E. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008, 2008.
URL: `https://doi.org/10.1088/1742-5468/2008/10/P10008`.

[5] Boldi, P., Codenotti, B., Santini, M., and Vigna, S. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice and Experience*, 34(8):711–726, 2004.
URL: `https://sparse.tamu.edu/GAP`.

[6] Borgatti, S. P. Centrality and network flow. *Social networks*, 27(1):55–71, 2005.
URL: `https://doi.org/10.1016/j.socnet.2004.11.008`.

[7] Brandes, U., Delling, D., Gaertler, M., Görke, R., Hoefer, M., Nikoloski, Z., and Wagner, D. On modularity - np-completeness and beyond. *ITI Wagner, Faculty of Informatics, Universität Karlsruhe (TH), Tech. Rep*, 19:2006, 2006.
URL: `https://i11www.iti.kit.edu/extra/publications/bdgghnw-omnpcb-06.pdf`.

[8] Britton, T., Deijfen, M., and Martin-Löf, A. Generating simple random graphs with prescribed degree distribution. *Journal of statistical physics*, 124:1377–1397, 2006.
URL: `https://doi.org/10.1007/s10955-006-9168-x`.

[9] Clauset, A., Newman, M. E. J., and Moore, C. Finding community structure in very large networks. *Physical review E*, 70(6):066111, 2004.
URL: `https://doi.org/10.1103/PhysRevE.70.066111`.

[10] Coffman, E. G., Elphick, M., and Shoshani, A. System deadlocks. *ACM Computing Surveys (CSUR)*, 3(2):67–78, 1971.
URL: `https://doi.org/10.1145/356586.356588`.

[11] Dijkstra, E. W. Hierarchical ordering of sequential processes. *Acta informatica*, 1: 115–138, 1971.
URL: `https://doi.org/10.1007/BF00289519`.

[12] Duch, J. and Arenas, A. Community detection in complex networks using extremal optimization. *Physical review E*, 72(2):027104, 2005.
URL: `https://doi.org/10.1103/PhysRevE.72.027104`.

[13] Euler, L. Solutio problematis ad geometriam situs pertinentis. *Commentarii academiae scientiarum Petropolitanae*, pages 128–140, 1741.
URL: `https://scholarlycommons.pacific.edu/euler-works/53`.

[14] eX3. Experimental infrastructure for exploration of exascale computing.
URL: `https://www.ex3.simula.no/`.

[15] Fortunato, S. and Barthelemy, M. Resolution limit in community detection. *Proceedings of the national academy of sciences*, 104(1):36–41, 2007.
URL: `https://doi.org/10.1073/pnas.0605965104`.

[16] Guimera, R. and Nunes Amaral, L. A. Functional cartography of complex metabolic networks. *nature*, 433(7028):895–900, 2005.
URL: `https://doi.org/10.1038/nature03288`.

[17] Haas, P. J. Data-stream sampling: Basic techniques and results. *Data Stream Management: Processing High-Speed Data Streams*, pages 13–44, 2016.
URL: `https://doi.org/10.1007/978-3-540-28608-0_2`.

[18] Hofmeister, M. Spectral radius and degree sequence. *Mathematische Nachrichten*, 139(1):37–44, 1988.
URL: `https://doi.org/10.1002/mana.19881390105`.

[19] Khalaf, E. G. and Tucci, R. Semi-contiguous memory allocation for efficient sequential-access. In *CDES*, pages 135–140, 2006.
URL: `https://www.researchgate.net/publication/220863115_Semi-Contiguous_Memory_Allocation_for_Efficient_Sequential-Access`.

[20] Klabnik, S. and Nichols, C. *The Rust programming language.* No Starch Press, 2023.
URL: `https://books.google.no/books?id=SE2GEAAAQBAJ&pg=PR`.

[21] Leskovec, J. and Mcauley, J. Learning to discover social circles in ego networks. *Advances in neural information processing systems*, 25, 2012.
URL: `https://snap.stanford.edu/data/ego-Twitter.html`.

[22] Lewis, T. G. *Network science: Theory and applications.* John Wiley & Sons, 2011.
URL: `https://books.google.no/books?id=eVddjxBhLsoC&pg=PT15`.

[23] Naim, M., Manne, F., Halappanavar, M., and Tumeo, A. Community detection on the gpu. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 625–634. IEEE, 2017.
URL: `https://doi.org/10.1109/IPDPS.2017.16`.

[24] Netzer, R. H. B. and Miller, B. P. What are race conditions? Some issues and formalizations. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1 (1):74–88, 1992.
URL: `https://doi.org/10.1145/130616.130623`.

[25] Newman, M. E. J. Finding community structure in networks using the eigenvectors of matrices. *Physical review E*, 74(3):036104, 2006.
URL: `https://doi.org/10.1103/PhysRevE.74.036104`.

[26] Newman, M. E. J. and Girvan, M. Finding and evaluating community structure in networks. *Physical review E*, 69(2):026113, 2004.
URL: `https://doi.org/10.1103/PhysRevE.69.026113`.

[27] Nguyen, F. Leiden-based parallel community detection. Master's thesis, Karlsruhe Institute of Technology, 2021.
URL: `https://i11www.iti.kit.edu/_media/teaching/theses/ba-nguyen-21.pdf`.

[28] Stroustrup, B. An overview of the c++ programming language. *Handbook of object technology*, page 72, 1999.
URL: `https://www.stroustrup.com/crc.pdf`.

[29] Traag, V. A., Waltman, L., and van Eck, N. J. From louvain to leiden: guaranteeing well-connected communities. *Scientific reports*, 9(1):5233, 2019.
URL: `https://doi.org/10.1038/s41598-019-41695-z`.

[30] Traag, V. A., Van Dooren, P., and Nesterov, Y. Narrow scope for resolution-limit-free community detection. *Physical Review E*, 84(1):016114, 2011.
URL: `https://arxiv.org/pdf/1104.3083.pdf`.

[31] Tumanis, A. Graph clustering for long term twitter observations community detection in incremental graphs. Master's thesis, University of Oslo, 2021.
URL: `https://www.duo.uio.no/bitstream/handle/10852/86015/1/UiO_IFI_Master_Thesis_Aigars_Tumanis.pdf`.

[32] Verweij, G. Faster community detection without loss of quality: Parallelizing the leiden algorithm. Master's thesis, Leiden University, 2019.
URL: `https://theses.liacs.nl/pdf/2019-2020-VerweijGeerten.pdf`.

[33] Watts, D. J. and Strogatz, S. H. Collective dynamics of 'small-world'networks. *nature*, 393(6684):440–442, 1998.
URL: `https://doi.org/10.1038/30918`.

[34] Wejdenstål, J. Dashmap.
URL: `https://docs.rs/dashmap/latest/dashmap/`.

[35] Zeng, J. and Yu, H. A distributed infomap algorithm for scalable and high-quality community detection. In *Proceedings of the 47th International Conference on Parallel Processing*, pages 1–11, 2018.
URL: `https://doi.org/10.1145/3225058.3225137`.